ACORN COMPUTER

The Concise BBC Microcomputer

USER GUIDE

RetroClinic.com
Vintage Arcade & Computing

ACORN COMPUTER

The Concise BBC Microcomputer
USER GUIDE

Vintage Arcade & Computing

The Concise BBC Microcomputer User Guide

Excerpts from original material written by John Coll, David Allen, Andrew Bray, Adrian Dickens, Mark Holmes.
1984 Amendments and corrections to this edition by Acorn Computers Limited.
2010 Amendments and typeset by RetroClinic

Part no 0433 000-01
Issue 1.2010.01.02
Date March 2010

WARNING: THE COMPUTER MUST BE EARTHED

IMPORTANT: The wires in the mains lead to the computer are coloured in accordance with the following code:

Green and yellow	**Earth**
Blue	**Neutral**
Brown	**Live**

As the colours of the wires may not correspond with the coloured markings identifying the terminals in your plug, proceed as follows:

The wire which is coloured green and yellow must be connected to the terminal in the plug which is marked by the letter E, or by the safety earth symbol or coloured green, or green and yellow.

The wire which is coloured blue must be connected to the terminal which is marked with the letter N, or coloured black.

The wire which is coloured brown must be connected to the terminal which is marked with the letter L, or coloured red.

If the socket outlet available is not suitable for the plug supplied, the plug should be cut off and the appropriate plug fitted and wired as previously noted. The moulded plug which was cut off should be disposed of as it could be a potential shock hazard if it were to be plugged in with the cut off end of the mains cord exposed. The moulded plug must be used with the fuse and fuse carrier firmly in place. The fuse carrier is of the same basic colour* as the coloured insert in the base of the plug. Different manufacturers' plugs and fuse carriers are not interchangeable. In the event of loss of the fuse carrier, the moulded plug MUST NOT be used. Either replace the moulded plug with another conventional plug wired as previously described, or obtain a replacement fuse carrier from an authorised BBC Microcomputer dealer. In the event of the fuse blowing it should be replaced, after clearing any faults, with a 3 amp fuse that is ASTA approved to BS1362.

*Not necessarily the same shade of that colour.

Exposure

Like all electronic equipment, the BBC Microcomputer should not be exposed to direct sunlight or moisture for long periods.

Econet and The Tube are trademarks of Acorn Computers Limited.

Original publication ©The author and the British Broadcasting Corporation 1982

Typeset and amendments © 2010 RetroClinic.

Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written approval of RetroClinic.

The product described in this manual and products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by Acorn Computers and RetroClinic in good faith. However, it is acknowledged that there may be errors or omissions in this manual. A list of details of any amendments or revisions to this manual can be obtained upon request from RetroClinic. Any suggestions and comments relating to the product and this manual are more than welcome.

All correspondence should be addressed to:

mark@retroclinic.com
http://www.retroclinic.com

All maintenance and service on the product must be carried out by Acorn Computers' authorised dealers. Acorn Computers and RetroClinic can accept no liability whatsoever for any loss or damage whatsoever caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers and RetroClinic shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

NOTE: If this manual is to be used in conjunction with a BBC Microcomputer which is fitted with an Operating System with version number lower than 2.00, then all references to the 'shadow screen' should be ignored.

In all other respects this manual is functionally compatible with earlier versions of the BBC Microcomputer.

Contents

Introduction 8
1. Getting started 10
The keyboard 13
2. Commands 18
3. A simple program 21
Using the screen editor 23
Deleting part of a program 26
Removing a program 28
Saving and loading a program 28
4. Sample programs 33
5. BASIC keywords 46
6. Error messages 229
Table of Error numbers 239
7. VDU drivers 241
Detailed description 243
8. Cassette files 255
Saving a BASIC program 256
Saving a section of memory 257
Loading a BASIC program 258
Loading a machine code program 259
Loading and running a BASIC program 259
Loading and running a machine code program 260
Using a cassette file to provide keyboard input 260
Reading cassette data files 261
Storing data on tape 262
Changing filing systems 267
9. Indirection operators 268
10. Operating System commands 274
11. Machine code and the BASIC Assembler 284
The BASIC Assembler 286
The assembler delimiters '[' and ']' 288
OPT - assembler option selection 289
The Location Counter P% 290

Labels 290
Forward Referencing and Two Pass Assembly 291
The EQUate Facility in Level 2 BASIC 291
Handling errors with BRK 293
Entering machine code from BASIC - CALL and USR 294
Conditional Assembly and Macros 294
User Zero Page 296
12. The 6502 Instruction Set 297
The 6502 registers and abbreviations 297
The Assembler Mnemonics 299
13. Operating System calls 358
14. *FX and OSBYTE calls 364
OSBYTE/*FX Call Summary 366
Detailed description of OSBYTE calls 371
15. OSWORD calls 432
16. Vectors 439
17. Memory usage 450
Zero page 450
Page one, &100-&1FF 455
Page two, &200-&2FF 455
Page three, &300-&3FF 457
Pages four through seven, &400-&7FF 461
Page eight, &800-&8FF 462
Page nine, &900-&9FF 462
Page ten, &A00-&AFF 463
Page eleven, &B00-&BFF 463
Page twelve, &C00-&CFF 463
Page thirteen, &D00-&DFF 463
The remainder of RAM, &E00-&7FFF 463
The ROMs 464
18. Events and Event handling 468
Event Descriptions 471
19. Interrupts 476
Non Maskable Interrupts 476
Maskable Interrupts 477
Interrupts on the BBC microcomputer 478

Serial interrupt processing 480
System VIA interrupt processing 482
User VIA interrupt processing 484
Intercepting interrupts 484
20. Paged ROMs 486
Entering Paged ROMs 488
Languages 494
Filing systems 496
ROM filing system software 497
21. Filing systems 499
The Cassette filing system 501
ROM filing system 504
Disc filing system 505
Advanced disc filing system 505
Econet filing system 506
RetroClinic RamFS 506
22. Using printers 507
23. The Hardware 513
SHEILA and the system hardware 516
The 6845 CRTC 517
The video ULA 533
The serial system 539
Paged ROM select register 546
The 6522 Versatile Interface Adapters 547
The System VIA 564
The 76489 sound chip 566
The User/Printer VIA 570
Floppy Disc and Econet 572
The Analogue to Digital converter 573
The One Megahertz bus 576
The Tube 587
Appendix 590
Appendix A - *FX/OSBYTE call index 590
Appendix B - Operating System calls summary 595
Appendix C - Key Values Summary 596
Appendix D - VDU Code Summary 599

Appendix E - Plot Number Summary 600
Appendix F - Screen mode layouts 601
Appendix G - Link options 618
Appendix H - The keyboard circuit 624
Appendix I - Glossary 626
Appendix J - The shadow screen 631
Appendix K - RetroClinic upgrades 634
Compact Flash Hard Drive 634
DataCentre USB System 638

Introduction

Equipment required

Before you start using your computer check that you have received the correct items in your package that you ordered. If your machine was specially built with additional items added internally, then you can check those using the separate manuals provided.

If you think you are short of any items, then contact your supplier by eMail, and request clarification that you have received the correct items.

Text conventions used in this manual

You will notice that the style of printing used to present the text in this manual varies. This is to help you tell the difference between explanatory text, words which appear on your monitor screen (including **BASIC** keywords) and certain keys on the computer keyboard.

- Ordinary text appears like this, or **like this** for emphasis.
- Text displayed on the screen (including **BASIC** keywords) appears `like this`.
- Words like **<RETURN>** mean that you should press the key marked RETURN rather than type the letters R E T U R N.

What this User Guide can and can't do

The BBC Microcomputer is a very versatile machine. On its own, connected to your television set or monitor, the computer can respond to programs which you yourself type in, to produce numbers, words, lines and movement on the screen and sound. Connect a suitable storage medium, such as cassette tape recorder, floppy disc, Compact Flash hard drive or USB, and you can then save your own programs for future use or run programs which have been written by other people. The WELCOME suite contains a number of programs specially written for the Machine. Other programs are available in large numbers, including programs linked to hobbies and games, and programs for use in the home, in business and in education. Languages other than **BASIC** (such as **LISP**, **FORTH**, **BCPL** and **PASCAL**) are available. These languages are stored in an integrated circuit which has to be plugged into your BBC Microcomputer. This must be done by your dealer.

The early chapters of this book will show you how to load and save, how to write simple programs and how to create certain graphics effects on the screen. There are also some complete programs to type in yourself. However, this is not a step-by-step course in **BASIC** programming. The tutorial sections of the original User Guide have been removed to make way for the later technical reference. If you are looking to learn **BASIC**, there are many good books available on the subject.

Most of what follows in the later chapters forms a reference guide on how to use the various commands and keywords of the BBC **BASIC** language. If you are an absolute beginner then much of this will not be very easy to understand. However, as you get more experience of programming, this material will prove invaluable.

1. Getting started

To get your computer working you will need a screen or some description. Most people at home will a domestic television, be that an early CRT type, or more modern LCD or plasma display. You will also need a storage device of some description if you wish to save and load programs., be that a cassette recorder, floppy disc, Compact Flash hard drive or USB interface..

If you have a high quality monitor, then it can be connected directly to one of the sockets at the back of the computer. To connect the monitor to the computer you will need a special monitor lead.

Assuming that you want to use an analog television set, then you can connect it to the computer using an aerial lead (if one was supplied with the computer). One of the plugs on this lead has a long central prong which fits into the socket on the back of the computer marked UHF out. The other end of the lead goes into the back of your television set in place of the normal aerial lead.

Next, plug your computer into the mains and switch it on (the On/Off switch is at the back). It should make a short 'beep' and the red light marked caps lock should come on. Turn the television on too and let it warm up for a moment.

Probably all you will see on the TV screen at this stage is a 'snow storm'. You will have to tune the TV so that it can receive the transmissions from the BBC Microcomputer. When your television is tuned correctly words will appear on the screen.

A message similar to:

```
BBC Computer 32K

Acorn 1770 DFS

BASIC

>_
```

should appear, which should be clear and sharp. Many types of tuning control indicate the channel number that you are tuning to. The BBC Microcomputer transmits on UHF channel 36, which is close to the frequency of Channel Five. It will not be too difficult to find the right channel but you will have to tune the TV carefully to get a really clear picture.

When you have a clear picture, do by all means press every button in sight on the computer – you can't do it any harm at all. Usually it just keeps on saying:

```
Mistake

>_
```

whenever you press the large key marked **<RETURN>**. `Mistake` just means that the computer does not understand your commands. Its fault – not yours!

You will see that if you hold any key down for more than a short time the character on the key appears on the screen, then there is a short pause, then the character repeats until you take your finger off again. On the whole, when pressing keys on the keyboard you should press them briefly – unless you want this repetition.

Experimenting

Now you are ready to experiment. You might like to try some of the following to see what the computer can do, but first be sure to press the key marked **<BREAK>**. This will clear the screen and get the computer ready for you.

Type in the following exactly as shown:

`MODE 5` **<RETURN>**

As you will see the command `MODE 5` clears the screen and just leaves the `>` mark on the screen. `>` is known as the 'prompt' and it means that the computer is ready for your next command.

Pressing the **<RETURN>** key tells the computer that you have finished the line you are typing and that you want it to obey your command. Before you press the **<RETURN>** key you can correct errors by pressing the key marked **<DELETE>**.

If the computer says `Mistake` then press the **<BREAK>** key and try again, starting with `MODE 5`.

Then type in each of the following lines – but don't forget to press the **<RETURN>** key at the end of every line. Don't worry if you make a mistake – it really doesn't matter!

```
DRAW 1000,100
DRAW 0,750
GCOL 0,1
PLOT 85,0,0
```

If the computer says `No such variable` then you are probably pressing the letter O instead of the number 0.

```
PLOT 86,1000,750
VDU 19,1,4,0,0,0
VDU 19,3,2,0,0,0
VDU 19,0,1,0,0,0
DRAW 200,0
DRAW 0,200
```

As you can see, the `DRAW` command is used to draw lines while `PLOT 85` and `PLOT 86` are used to plot and fill in triangles on the screen. When using the graphics the points on the screen are numbered from 0 to 1279 (left to right) and from 0 to 1023 (bottom to top). They are like positions on a piece of graph paper.

Words can also be plotted in colours, as you will have seen. Clear the screen by typing `MODE 5` and then type the following:

`COLOUR 1`	This selects a red foreground.
`COLOUR 2`	This selects a yellow foreground.
`COLOUR 3`	This selects a white foreground.
`COLOUR 129`	This selects a red background.
`COLOUR 0`	This selects a black foreground.
`COLOUR 130`	This selects a yellow background.

The computer can create sound as well. Try typing this in:

```
SOUND 1,-15,100,200 <RETURN>
```

That gives a simple, crude sound. It is also possible to alter the quality of the sound. Try this:

```
ENVELOPE 2,3,2,-4,4,50,50,50,127,0,0,0,126,0
```

(This should be typed in as one line even though it may spill over to the next line on the screen. The computer will treat it as being 'one line' when you press <RETURN>.) Now carry on with:

```
SOUND 1,2,1,10
SOUND 2,2,100,1
SOUND 3,2,200,1
```

You will have to press <ESCAPE> to stop the sound.

Here's another one:

```
ENVELOPE 1,1,-26,-36,-45,255,255,255,127,0,0,0,126,0
SOUND 1,1,1,1
```

Take a look at a more detailed description of the SOUND command in Chapter 5.

The keyboard

Anyone who has used a standard typewriter will be familiar with the positions of most of the symbols on the keyboard of the BBC Microcomputer. However, there are a number of special keys which need to be mastered (see Figure 1.1) and these are described below.

If you are a keyboard novice you may find the layout confusing. Don't worry – first of all it is not necessary to be a touch typist to work the computer; secondly, there is a program in the WELCOME suite which will help you to practice finding the various keys, and most people find that with a little practice they become familiar with them fairly quickly.

Some keys have two symbols engraved on them – we'll call those on the top

'upper case' and those below 'lower case' symbols.

CAPS LOCK

When the machine is switched on, the middle light should be on, telling you that the **CAPS LOCK** key is on. This gives capital letters and lower case symbols and is the most useful state for programming because **BASIC** only recognises commands typed in using capital letters. By pressing the <CAPS LOCK> key once you can switch the light off. Now you get lower case letters and lower case symbols. Press it again and it will be on again.

SHIFT

Whether **CAPS LOCK** is on or off, if you press either of the <SHIFT> keys and hold it down while typing in a character you will get a capital letter or upper case symbol.

Holding down <CTRL> and <SHIFT> together stops the computer 'writing' to the screen. This can be useful if it is writing faster than you can read.

SHIFT LOCK

Pressing this key once gives capital letters and upper case symbols until it is pressed again. It has its own on/off light.

Practice in the use of these keys is given in one of the first programs in the introductory pack – the one called KEYBOARD.

SHIFT CAPS LOCK

Depressing and releasing <SHIFT> and <CAPS LOCK> in unison reverses the effect of the <SHIFT> key, causing a lower case letter to be printed when pressing a letter key with <SHIFT> held down. Pressing <CAPS LOCK> again returns the keyboard to normal.

RETURN

This key is the most commonly used key on the keyboard. When a command or anything else is typed in, it is not usually acted upon until the <RETURN> key is pressed. In other words, this key informs the computer that you have finished entering a line or a reply. Until you press <RETURN>, you can add to or delete what you have typed in.

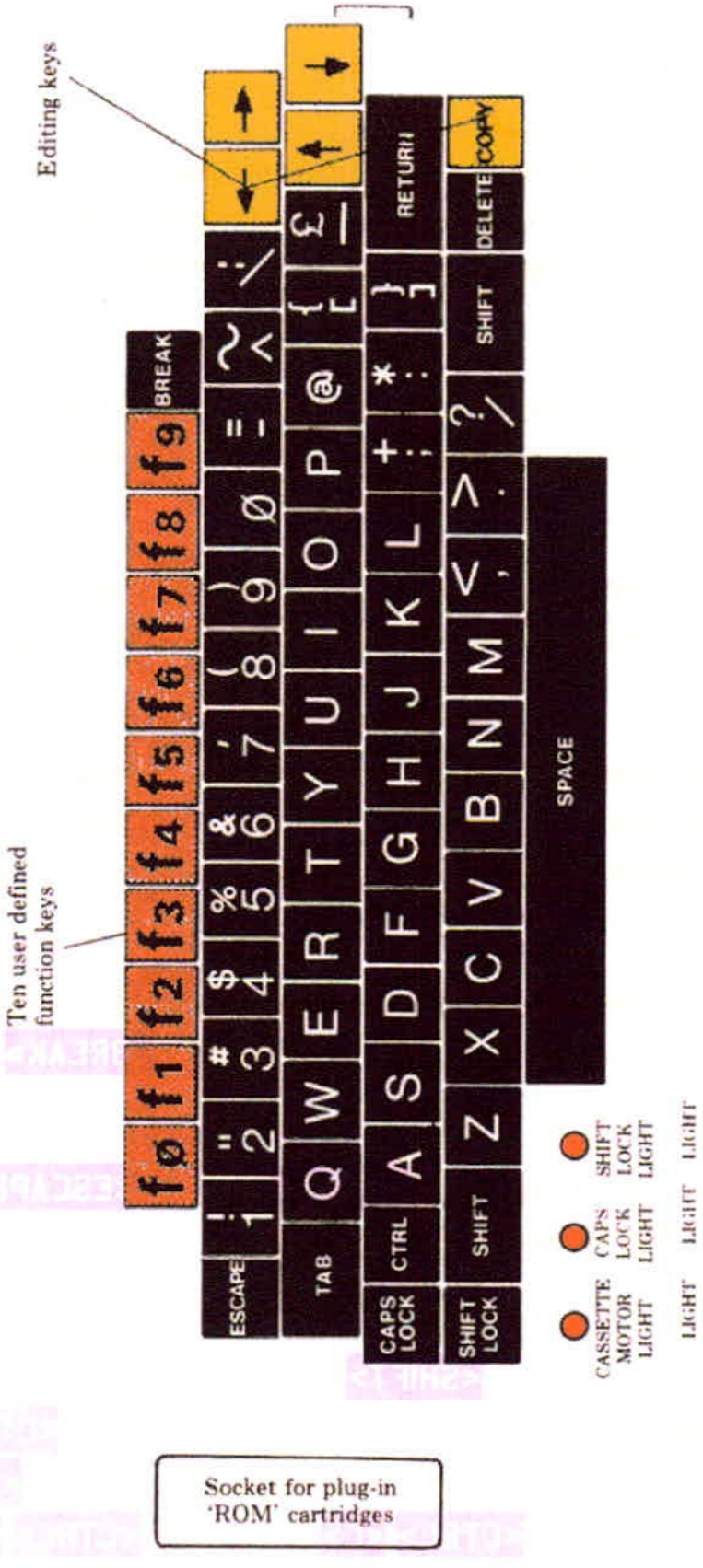
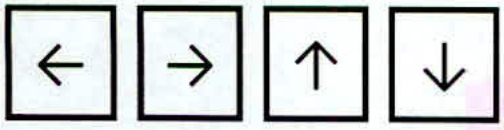


Figure 1.1 – Keyboard Layout

CURSOR control keys



These enable you to move the flashing cursor around the screen when editing a program. Pressing any of them makes the computer automatically enter the 'editing mode' during which two cursors are shown on the screen.

DELETE

Pressing this key will cause the last character typed in to be erased from the screen. If held down, it will then erase further characters until released.

COPY

This key, used in conjunction with the cursor control keys, enables anything on the screen to be copied – a useful feature when editing a line in a program.

ESCAPE

This key is usually used to stop a program which is running. However, it can be programmed to do other things when pressed – such as moving you from one part of a program to another.

BREAK

This key stops the computer no matter what it is doing. The computer forgets almost everything that it has been set to do. Pressing <BREAK> also resets the screen to Mode 7.

Do not get into the habit of using the <BREAK> key. The <ESCAPE> key provides a much less violent way of escaping from a program!

CTRL

This key behaves similarly to the <SHIFT> key in that it can be used to change the character generated by other keys. For example, pressing <CTRL> and <G> (called Control G) makes the internal speaker make a short noise. <CTRL> <B> is used to turn a printer on and <CTRL> <C> turns it off. <CTRL> <N> makes the computer stop at the bottom of each page, etc, etc. More information on control codes is given in Chapter 6 – VDU Drivers.

TAB

Another key that is useful in special circumstances – like word processing.

These keys can be somewhat confusing because they seem to generate the wrong characters sometimes. The problem is that there are two international standards for displayed characters (Teletext and ASCII) and the BBC Microcomputer can display either. Mode 7 generates the Teletext display characters and Modes 0 to 6 show the ASCII characters. But don't worry, the computer recognises the key correctly regardless of what is displayed on the screen. Here is a table showing all these characters:

On the key	Displayed on the screen	
	In Mode 7	**in Modes 0 to 6**
~	÷	~
^	↑	^
|	|	|
\	½	\
{	¼	{
[	←	[
}	¾	}
]	→	]

Note that in Mode 7 a zero is shown as a rather pointed O whereas in all other modes, zeros have a slash – 0 – to help to differentiate them from the letter O. The keyboard is also marked in this way.

2. Commands

There are two ways of getting the computer to do something:

1. Give it commands which it can act on immediately. This is what happened when you typed in the lines in chapter 1.
2. Give it a series of numbered instructions, often called statements, which it can store in its memory and carry out in sequence when told to do so. A stored series of instructions is called a program.

Many of the keywords in **BASIC** can be used both as commands and as statements in a program.

The rest of this chapter is concerned with 'command mode'.

`PRINT` is used to make the computer print something on the screen. Try these two examples:

```
PRINT "HELLO"
```

Don't forget to press **<RETURN>** at the end of each line.

```
PRINT 3+4
```

In the second example you have given the computer a command to print the sum of 3 and 4. The computer can very easily do addition, subtraction, multiplication and division. The addition, subtraction, multiplication and division signs are all on the right side of the keyboard. If you are interested in doing mathematical or financial work then you will need to know the symbols that the computer uses for various mathematical operations. They are:

+	Addition
-	Subtraction
*	Multiplication
\	Division
^	Exponentiation
.	Decimal point

If you want to get the + or * then you will have to press the <SHIFT> key as well as the key you want. It's rather like a typewriter: while holding the <SHIFT> key down, press the + sign once.

Try typing in the following and check that they work – in other words see that they produce the expected answers:

```
PRINT 4+8
PRINT 18-2*4
PRINT 131/4
PRINT SQR(2)
```

The last one will print the square root of 2 which is 1.41421356.

Then try:

```
MODE 5
```

which will make the computer clear the screen and get it ready to draw lines as well as text. In this mode:

```
COLOUR 129
```

will select a red background, and:

```
CLS
```

will clear the screen to the background colour. In each case you have given the computer a command and it has obeyed it immediately. Working like this is called 'working in command mode'.

While in this mode you might like to learn how to use the bright red *user defined function keys*. Each of these keys can be used to store a word or several words. For example they could be programmed so that each one selects a different colour. Try this:

```
*KEY 2 COLOUR 2|M
```

The | shown above is produced by a special key. On the keyboard this key is the third key from the right on the row below the red keys.

Once you have typed that in then every time you press the key marked <F2>, the computer will change to COLOUR 2 which gives yellow lettering. In a similar way you could program some of the other keys like this:

```
*KEY 0 COLOUR 0|M
*KEY 1 COLOUR 1|M
*KEY 3 COLOUR 3|M
```

Note the exact position of spaces when you type in a command.

Of course red letters don't show up very well on a red background! You will have noticed the |M at the end of each line above. That is the code used to get a <RETURN> into the user defined function keys.

If the picture on your television screen is either too far up or too far down the screen, you can move the whole display with the command *TV.

*TV 255	will move down one line
*TV 254	will move down two lines
*TV 1	will move up one line
*TV 2	will move up two lines

The movements come into effect next time you press <BREAK> or change mode.

*TV also controls the interlace of the television display, and is further covered in Chapter 10.

3. A simple program

In the previous chapter we have been giving the computer commands which it obeys immediately. The problem with this technique is that you have to wait until the computer has completed one command before you can give it the next one. If the computer takes a long time to work out one of the problems you have set it, then you may have to waste an awful lot of time just sitting there waiting for it. For example if you want your computer to work out the number of £5, £10 and £20 notes that you will need to pay the wages at the end of the week the computer will take a fair time to calculate all the wages before it can sort out the notes required.

The same problem comes up when you take a car into a garage to be serviced. You could for example stand by the mechanic and say 'First of all I want the oil changed' and then you could wait for him to change the oil. When that is completed you could then say 'Now I want you to replace the bulb that has blown in one of the front headlights' and then you could wait for that job to be done. And thirdly you might say 'The exhaust is making a noise, so I want you to put the car up on the ramp and check it'.

You would spend a great deal of time waiting for the mechanic to complete each job before assigning the next. There is a far more efficient way of doing things; when you go into the garage you give the mechanic a whole set of instructions, for example:

- First of all change the oil.
- Secondly replace the headlight bulb.
- Thirdly stop the noise in the exhaust.

Once you have given your set of instructions and checked that the garage understands what has to be done, you can walk off and have a cup of coffee and then go back expecting the job to be finished. Now the same thing applies with a computer. It is far better to give it a whole set of instructions and let it run while you wander off and have a cup of coffee. 'Writing a computer program' is nothing more than giving a set of instructions.

If you give the computer a command like:

```
PRINT "HOW ARE YOU"
```

then the computer will do that immediately. On the other hand, if you give the computer a statement:

```
10 PRINT "HOW ARE YOU"
```

then the computer will regard that as instruction number 10 and it will not do it immediately, but expect other instructions to follow. Instruction number 10 is usually referred to as line 10. So, if there is a line number then the statement is part of a program; if there is no line number then it is a command which the computer must obey immediately.

When you have given the computer a set of instructions and you then want it to carry them out, you type the word **RUN** on the keyboard. The computer will then carry out the instructions that you asked it to do one at a time and in line number order. In other words, it will 'execute' the program that you have typed in. Just to check that you have got the idea of what is going on, here is a small program that you can type in.

```
10 REPEAT
20 PRINT "GIVE ME A NUMBER";
30 INPUT B
40 PRINT "12 TIMES ";B;" IS ";12*B
50 UNTIL B=0
```

When you **RUN** the program line 20 will print the message:

```
GIVE ME A NUMBER
```

on the screen.

Line 30 will print a question mark on the screen and wait for you to type in a number (followed by **<RETURN>** – as usual). The number you type in will become the value of the variable 'B'.

Line 40 will first print the words `12 TIMES` followed on the same line by the number you typed in, followed on the same line by the word `IS` followed by the result of the calculation. The semi-colons tell the computer to print the next item on the same line as the previous one and right up against it.

Line 50 sends the computer back to line 10 unless B=0, when the program will stop.

Another way of stopping the program is to press the 'panic button' which is marked **ESCAPE** (it's at the top left of the keyboard). If the computer seems to be ignoring you because it's too busy running a program. You can nearly always get its attention by pressing the **<ESCAPE>** key. When you do that it will stop running your program and print a `>` prompt to show that it has stopped the program and is waiting for your command.

When the computer shows a `>` it is in command mode. You can change your program, give it commands for immediate execution, or tell it to `RUN` the program (in its memory) again. It doesn't forget a program when you press **<ESCAPE>**.

If the computer is in command mode (in other words if the last thing on the screen is `>`) then you can command it to print the program in its memory by typing:

`LIST` **<RETURN>**

The computer will then give a listing of the program on the screen for you to check. If you discover that you have made an error, for example that you have got something wrong in line 20, then it is easy to correct the error. There are two ways of correcting major errors:

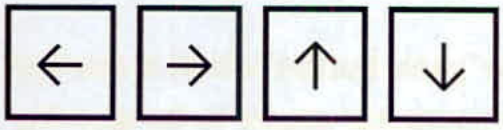
- Retype the whole line.
- Use the screen editor.

Using the screen editor

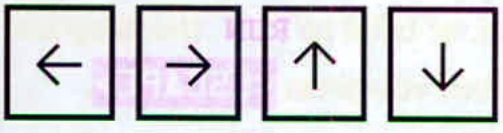
There is a group of six keys on the right hand side of the keyboard which can be used to edit, or alter, program lines that are displayed on the screen. Four of the keys have arrows on them and are coloured a lighter brown that most of the other keys. These keys enable you to move a flashing cursor around the screen to a line that you wish to edit. As soon as you press one of these keys the computer enters a

special 'editing mode' where it displays two cursors. The large white block is called the write cursor and it shows you where anything that you enter will appear.

The other small, flashing cursor – the read cursor – is the one that can be moved around by the arrow keys.



Try moving the read cursor, by using the arrow keys, until it is under a letter at the start of a word and then press the **<COPY>** key several times. As you will see the **<COPY>** key copies everything that the read cursor passes under into the new input line. Halfway through copying a line you can always use:



to move the read cursor to some new place on the screen before using **<COPY>** again to copy some other text to your new input line. The **<DELETE>** key can always be used to delete characters from the input line.

You can also type new characters in at any time instead of using the **<COPY>** key. When your new input line is complete just press **<RETURN>** in the usual way.

Try the following: clear the screen with the command `CLS` and then `LIST` the program. It should include the line:

```
20 PRINT "GIVE ME A NUMBER";
```

If not, then type that line in so that you can edit it. Suppose that you wanted to insert the word `BIG` so that line 20 reads:

```
20 PRINT "GIVE ME A BIG NUMBER";
```

then all you have to do is to press the up-arrow cursor key until the small flashing line is positioned under the `2` of `20`. Then press the **<COPY>** key to copy the first part of line 20 to a fresh line at the bottom. When the cursor reaches the space

after the A where you want to insert the word BIG, just type it in with a space in front – it will appear on the bottom line. Then <COPY> the rest of the line 20. The space after the A becomes the space after BIG. At the end press <RETURN>.

Now try changing the program already in the computer once again by doing the following:

1. List the program by using the LIST command.
2. Practice using the cursor control and <COPY> keys to alter line 20 so that it reads:

```
20 PRINT "NOW GIVE ME A BIG NUMBER";
```

3. Now add these new lines. Don't forget to press <RETURN> after each one:

```
5 CLS
25 REPEAT
35 IF B<1000 THEN PRINT "I SAID A BIG NUMBER"
37 UNTIL B>=1000
```

NOTE: It doesn't matter in what order you type new lines. The computer will automatically put them into numerical order. You will see that this is true by typing:

```
LIST <RETURN>
```

These extra lines tell the computer to reject any number smaller than 1000 and to keep on going back to line 30 to ask for a new number until that number is greater than 1000. The symbol < means 'is smaller than', and > means 'is greater than'. IF and THEN are self explanatory.

4. Now RUN the program.

```
>RUN
NOW GIVE ME A BIG NUMBER? 16
I SAID A BIG NUMBER
?20
I SAID A BIG NUMBER
?2000
```

```
12 TIMES 2000 IS 24000
NOW GIVE ME A BIG NUMBER?
```

This program will go on running until you press **<ESCAPE>**. If you look you will see that if you give the value 0 for the number, the program never reaches line 50, so it can never end unless you press the panic button!

Deleting part of a program

Quite often you will want to delete a whole line or group of lines in your program. This is easy to do but don't forget that if you type in a new line 20 (for example), it will automatically remove the old line 20 and replace it with your new one. If you want to delete a line completely then type in just the line number and press **<RETURN>**:

```
20 <RETURN>
```

To delete a whole set of line numbers, for example, lines 50 to 70 inclusive, you can type:

```
DELETE 50,70
```

You cannot get these lines back once they are deleted – unless you can copy them off the screen, so use this with care.

After you have deleted several lines – or if you have typed in lots of new lines you often find that you have a very odd set of line numbers. The command:

```
RENUMBER
```

will make the computer go through your whole program renumbering all the lines so that they are given line numbers in a numeric sequence. Here is an example of terrible programming style – but it will illustrate the **RENUMBER** command. Don't bother to type it in – just look at it.

```
>LIST
  1 REM ** GOTO GOTO GOTO
  2 REM WITH ACKNOWLEDGEMENTS TO
  3 REM "COMPUTERS IN SCHOOLS"
  4 REM THE JOURNAL OF MUSE
 15 GOTO 100
 16 GOTO 95
 40 N=N+1
 44 END
 57 IF N=18 THEN PRINT "GOTO OR NOT TO GOTO"
 60 IF N>35 THEN GOTO 110
 78 GOTO 40
 95 PRINT "**THE GOTO SHOW**": GOTO 40
100 N=0: GOTO 16
105 PRINT "GOT TO GOTO GOTO NOW"
110 GOTO 44
115 PRINT "GOTO OR NOT TO GOTO";:GOTO 60
>RENUMBER
>LIST
 10 REM ** GOTO GOTO GOTO
 20 REM WITH ACKNOWLEDGEMENTS TO
 30 REM "COMPUTERS IN SCHOOLS"
 40 REM THE JOURNAL OF MUSE
 50 GOTO 130
 60 GOTO 120
 70 N=N+1
 80 END
 90 IF N=18 THEN PRINT "GOTO OR NOT TO GOTO"
100 IF N>35 THEN GOTO 150
110 GOTO 70
120 PRINT "**THE GOTO SHOW**": GOTO 70
130 N=0: GOTO 60
140 PRINT "GOT TO GOTO GOTO NOW"
150 GOTO 80
160 PRINT "GOTO OR NOT TO GOTO";:GOTO 100
>RUN
**THE GOTO SHOW**
```

As you will see, the **RENUMBER** command has not only renumbered the references to line numbers which occur within the program itself – namely after the statements containing the keyword **GOTO**. (This gives the computer the instruction to go to a particular line number and carry out the instruction it finds there.)

Removing a program

If you want to write a new program you will want to remove the old program from the computer's memory. This can be done by using the command `NEW`, or by pressing the <BREAK> key. In either case, if you regret having lost your program, type `OLD` and press <RETURN> and, providing you haven't begun to type in the new program, the old one should reappear.

You can always check what's in the memory by typing `LIST`. Try experimenting with these various commands on the program you have typed in.

Saving and loading a program

Programs can be saved and loaded to and from your computer using various methods. If you have a floppy disc drive, Compact Flash hard drive, or some other storage device, please refer to the instructions that came with that unit. In the next section, we will cover saving and loading from cassette.

You can store a copy of any program on cassette and then load it back into the machine at some time in the future. It really is just like recording music onto a cassette – you can then play the cassette back a few days later and the music will still be there.

If you decide that you don't want to keep the computer program that you have saved on cassette then you can just record a new program over the old one in the same way that you can re-use a cassette when recording music. And in the same way that it is very easy to forget where a particular piece of music is recorded on a cassette, so it's very easy to forget where on the cassette you have stored a particular program. It is very strongly suggested that you use the tape counter to keep an index of where programs are on cassette. Also you must leave gaps between programs. It is easy to let one program run over the start of the next one if they are all squashed close together. If programs do overlap then you will definitely lose one of them. Be warned!

Most short programs will only move the cassette tape counter on 30 or 40 positions but play safe and spread the programs out over the length of the cassette. If you record the first program at 0000, the second at 0100, the next at

0200 and so on then they will be easy to find and they are unlikely to run over each other.

NOTE: Don't make the mistake of trying to record on the clear plastic tape 'leader' – wind the tape on by hand until the brown tape itself is exposed.

Saving a program on cassette

If you have typed a program into your microcomputer then all you have to do to save it is to:

1. Insert the cassette into the recorder.
2. Set the tape counter to 0000 when the tape is fully re-wound.
3. Type

```
SAVE "MYPROG"  <RETURN>
```

4. The message `RECORD then RETURN` will appear.
5. Fast forward the cassette to the place where you want to record the program - this will be 100 or 200 or 300 etc, on the tape counter.
6. Press the RECORD button on the cassette and then press the **<RETURN>** key.

If you want to give up at any time then press the **<ESCAPE>** key.

Notice that `MYPROG` is the name that we happened to give to the program. You can call your program by any name you like so long as it has no more than ten characters. For example you could have typed:

```
SAVE "FRED"
SAVE "GAME3"
SAVE "picture"
```

While the program is being saved on cassette the name of the program and some numbers will appear to tell you that things are happening. When the computer has finished, the `>` prompt will re-appear and the tape will stop automatically. If you don't have cassette motor control then you will have to stop the recorder manually after the `>` prompt re-appears. That's it!

Checking a recording

If you want to check that you have successfully recorded your program on the tape then you can use the `*CAT` command (see below). If your recording failed for any reason you can always re-record it.

Loading a program from cassette

Loading a program back into the computer is just like playing a particular piece of music which has been recorded on the cassette.

1. Type:

```
LOAD "MYPROG" <RETURN>
```

The message:

```
Searching
```

will appear. Of course if your program is called something else then use the right name, for example:

```
LOAD "GAME3" <RETURN>
```

2. Rewind the cassette to just before the start of your program (which will be at 100 or 200 etc.)
3. Check that the volume and tone control settings are correct.
4. Start playing the cassette by pressing the PLAY button on the recorder.

When the computer finds any program on the cassette it will show the name of the program on the screen. When it finds the program it is looking for it will print:

```
Loading
```

to let you know that it is now loading the right program.

When the computer has finished loading the program it will print the > prompt. It will also automatically stop the tape if you have automatic motor control, if not then you will have to stop the tape manually.

The program is now in the computer. You can type:

```
RUN <RETURN>
```

to make it work, as usual.

There is one more useful feature to do with loading and saving programs. Instead of typing:

```
LOAD "MYPROG" <RETURN>
```

you can type:

```
CHAIN "MYPROG" <RETURN>
```

This not only loads in the program `MYPROG` but also starts it working as soon as it has loaded. It is normally more convenient to use `CHAIN` than `LOAD`.

Cataloguing a tape

If you forget what programs you have on the tape then you can get a catalogue by typing:

```
*CAT
```

and then playing the tape, but you'll have to wait until the tape has run through the programs.

What the numbers mean

A typical catalogue looks like this:

```
WELCOME     00 0084
INTRO       08 088E
INDEX       0A 0ABA
KEYBOARD    25 2545
```

The file-name is followed by two 'hexadecimal' numbers which give the 'block number'. Each program is recorded as a series of 'blocks'. The last number on the line gives the 'length' of the file.

The action of cataloguing a tape also lets the computer verify the information recorded. If there are errors in any of the data on the tape it will print a message and continue.

The **<ESCAPE>** key allows you to leave cassette operations whenever you like. If you leave from the middle of a **LOAD** operation you will probably get a:

```
Bad Program
```

error. Type **NEW** to remove this.

4. Sample programs

Chapter 5 lists all the commands available to you in **BASIC**. Before we do that, here are a few complete programs which you can try to type in yourself. They must be typed in accurately and can then be run. If a program fails to run properly, then you probably typed a line in incorrectly – for instance, you may have typed `;` when you should have typed `:` or typed `O` instead of `0`.

Most of the sample programs are too big to fit all of the lines on the screen. If you `LIST` a program you have typed in, for example to check that you have made no mistakes, you may find that the lines you want to look at disappear off the top of the screen. To prevent this you can specify the range of lines you want to be listed. For example:

```
LIST 100,200
```

will only list those lines numbered between 100 and 200.

Alternatively you can enter 'paged mode' by pressing <CTRL> <N> (hold down <CTRL> and press <N>). In this mode the listing will stop after every 'page' and will continue only when you press the <SHIFT> key. Paged mode is switched off by pressing <CTRL> <O> and you should always remember to do this after you have listed the program.

Typing in programs will help you to get a feel for the keyboard and, if you save them on cassette after you are satisfied that they do run properly, will enable you to start building up a library of them.

Learning to use the computer is a little like learning to drive a car – when you first start you find that there are an enormous number of things to think about all at one time. Many of the things you come across from now on will be bewildering at first, but as you get further into the book and as you gain experience in using **BASIC**, the various parts of the jigsaw puzzle should begin to fall into place. So don't worry if, for instance, some of the comments about the following programs are difficult to understand at first.

NOTE: In the program listings which follow, extra spaces have been inserted between the line numbers (10,20, etc) and what follows on each line. This is to improve the readability of the programs. However, although it will do no harm, there is no reason to type in any spaces after the line number. For example in the first program, called POLYGON, when entering line 250, all you need to type is:

```
250MOVE 0,0
```

POLYGON

This program draws polygons (many sided shapes) in random colours. Lines 120 to 180 move to a random place on the screen which will be the centre (origin) of the next shape. Lines 210 to 290 calculate the X and Y coordinates of each 'corner' of the polygon and store the values in two 'arrays' for future use. In addition the shape is filled with black triangles (lines 260 and 290) that make it appear that the new polygon is in front of the older ones. Lines 310 to 370 draw all the lines that make up the polygon.

Lines 50 to 70 set the actual colour of logical colours 1,2 and 3 to red, blue and yellow. You can change these if you wish to use other colours.

```
 10 REM POLYGON
 20 REM JOHN A COLL
 30 REM VERSION 1 / 16 NOV 81
 40 MODE5
 50 VDU 19,1,1,0,0,0
 60 VDU 19,2,4,0,0,0
 70 VDU 19,3,3,0,0,0
 80 DIM X(10)
 90 DIM Y(10)
100
110 FOR C=1 TO 2500
120 xorigin=RND(1200)
130 yorigin=RND(750)
140 VDU29,xorigin;yorigin;
150 radius=RND(300)+50
160 sides=RND(8)+2
170 MOVE radius,0
180 MOVE 10,10
190
200 GCOL 0,0
210 FOR SIDE=1 TO sides
220 angle=(SIDE-1)*2*PI/sides
```

```
230 X(SIDE)=radius*COS(angle)
240 Y(SIDE)=radius*SIN(angle)
250 MOVE0,0
260 PLOT 85,X(SIDE), Y(SIDE)
270 NEXT SIDE
280 MOVE0,0
290 PLOT 85,radius,0
300
310 GCOL 0,RND(3)
320 FOR SIDE=1 TO sides
330 FOR line=SIDE TO sides
340 MOVE X(SIDE), Y(SIDE)
350 DRAW X(line), Y(line)
360 NEXT line
370 NEXT SIDE
380 NEXT C
```

You may like to try this alternative for line 200

```
200 GCOL 0, RND(4)
```

MONTHLY

This program plots a set of 'blocks' on the screen which might represent prices over a 12-month period. In this example the height of the bars is randomly selected at line 170. Lines 180 to 270 then draw a 'solid' bar and the edges are marked in black by lines 290 to 330. Lines 340 and 350 print out one letter representing the month of the year at the bottom of each bar.

Notice that lines 60 and 70 set up two of the function keys. Key <f0> sets the computer to `MODE 7` and then lists the program. Key <f9> can be used to run the program.

```
 10 REM MONTHLY
 20 REM JOHN A COLL
 30 REM VERSION 1 / 16 NOV 81
 50
 60 *KEY 0 "MODE7|MLIST|M"
 70 *KEY 9 "RUN|M"
 80 M$="JFMAMJJASOND"
 90 C=0
100 MODE 2
110 VDU5
120 VDU 29,0;100;
130
```

```
140 FOR X=0 TO 1100 STEP 100
150 GCOL 0,C MOD 7+1
160 C=C+1
170 H=RND(400)+200
180 MOVE X,0
190 MOVE X,H
200 PLOT 85,X+100,0
210 PLOT 85,X+100,H
220 MOVE X+70,H+50
230 MOVE X,H
240 PLOT 85,X+170,H+50
250 PLOT 85,X+100,H
260 PLOT 85,X+170,50
270 PLOT 85,X+100,0
280 GCOL,0
290 MOVEX,H
300 DRAW X+100,H
310 DRAW X+170,H+50
320 MOVE X+100,H
330 DRAW X+100,0
340 MOVE X+10,50
350 PRINT MID$(M$,C,1)
360 NEXT
370
380 GCOL 4,1
390 MOVE 0,450:PRINT "----------------"
400 VDU4
410 PRINTTAB(3,0)"critical level"
```

The height of each bar is set randomly by the variable H. If you want to put in your own values (data), then type the following extra lines. Line 170 must also be deleted by typing:

```
170 <RETURN>
```

Now add the extra lines:

```
 50 DIM data(12)
 82 FOR J=1 TO 12
 84 PRINT "Input data for month "MID$(M$,J,1);
 86 INPUT data(J)
 88 NEXT J
 89 INPUT "CRITICAL LEVEL", level
155 H=data(C+1)
390 MOVE 0,level:PRINT"---------------"
```

QUADRAT

This program can be used to solve equations of the form:

$Y=Ax_2 + Bx + C$

The 'roots of the equation' are printed to two decimal places.

The number of decimal places is set by line 90.

The main program between lines 110 and 170 uses three procedures – one for each of the three types of result. The main program is surrounded by:

```
REPEAT
.....
.....
.....
UNTIL FALSE
```

which keeps the program going for ever – or until the <ESCAPE> key is pressed.

Line `170 PRINT'''` prints three blank lines to separate one set of results from the next.

```
 10 REM QUADRAT
 20 REM JOHN A COLL BASED ON A PROGRAM
 30 REM BY MAX BRAMER, OPEN UNIVERSITY
 40 REM VERSION 1.0 /16 NOV 81
 50 REM SOLVES AN EQUATION OF THE FORM
 60 REM A*X^2 + B*X + C
 70 ON ERROR GOTO 350
 80 MODE 7
 90 @%=2020A
100 REPEAT
110 PRINT "What are the three coefficients ";
120 INPUT A,B,C : IF A=0 THEN 110
130 DISCRIM=B^2-4*A*C
140 IF DISCRIM<0 THEN PROCcomplex
150 IF DISCRIM=0 THEN PROCcoincident
160 IF DISCRIM>0 THEN PROCreal
170 PRINT'''
180 UNTIL FALSE
190 END
200
210 DEF PROCcomplex
```

```
220 PRINT "Complex roots X=";-B/(2*A);
230 PRINT " +/- "; ABS(SQR(-DISCRIM)/(2*A)) "i"
240 ENDPROC
250
260 DEF PROCcoincident
270 PRINT "Co-incident roots X=";B/(2*A)
280 ENDPROC
290
300 DEF PROCreal
310 X1=(-B+SQR(DISCRIM))/(2*A)
320 X=2(-B-SQR(DISCRIM))/(2*A)
330 PRINT "Real distinct roots X=";X1;" and X=";X2
340 ENDPROC
350 @%=&90A:REPORT:PRINT" at line "ERL

>RUN
What are the three coefficients ?1,-1,-2
Real distinct roots X=2.00 and X=-1.00
What are the three coefficients ?3,3,3
Complex roots X=-0.50 +/- 0.87i
What are the three coefficients ?1,2,1
Co-incident roots X=1.00
What are the three coefficients ?
Escape at line 120
>
```

FOURPNT

This program draws a pattern (lines 80 to 140) and then changes foreground and background colours with a half second pause between each change.

```
 10 REM FOURPNT/DRAWS A PATTERN WITH 4 POINTS
 20 REM JOHN A COLL
 30 REM VERSION 1 /16 NOV 81
 50 MODE 4
 60 VDU 29,640;512
 70
 80 FOR A=0 TO 500 STEP 15
 90 MOVE A-500,0
100 DRAW 0,A
110 DRAW 500-A,0
120 DRAW 0,-A
130 DRAW A-500,0
140 NEXT A
150
160 FOR B=0 TO 7 : REM CHANGE THE COLOUR
170 FOR C=1 TO 3
180 T=TIME : REM WAIT A WHILE
```

```
190 REPEAT UNTIL TIME-T>50
200 VDU 19,3,C,0,0,0
210 VDU 19,0,B,0,0,0
220 NEXT C
230 NEXT B
```

TARTAN

This program builds up a changing pattern by overdrawing lines on the screen.

The main program between lines 90 and 140 loops for ever and calls various subroutines as necessary. The use of subroutines with implied **GOTO** (e.g. line 170) results in a structure which is not easy to follow! It would be better to use 'structures' such as procedures (see Chapter 5, the **PROC** command).

```
 10 REM TARTAN
 20 REM BASED ON RESEARCH MACHINES DEMO
 30 REM VERSION 1.0/16 NOV 81
 40 MODE 2: REM ALSO WORKS IN MODE 5
 50 R=1: D=1: X=0
 60 Y=RND(800)
 70 MOVE X,Y
 80
 90 REPEAT
100 ON D GOSUB 160,260,350,430
110 IF RND(1000)<10 THEN R=D-1
120 GCOL R,(D*1.7)
130 DRAW X,Y
140 UNTIL FALSE
150
160 X=X+800-Y
170 IF X>1000 THEN 220
180 Y=800
190 D=2
200 RETURN
210
220 Y=800/1000-X
230 X=1000: D=4
240 RETURN
250
260 Y=Y-800+X
270 IF Y<0 THEN 310
280 X=1000: D=3
290 RETURN
300
310 X=1000+Y
320 Y=0: D=1
```

```
330 RETURN
340
350 X=X-Y
360 IF X<0 THEN 400
370 Y=0: D=4
380 RETURN
390
400 Y=-X: X=0: D=2
410 RETURN
420
430 Y=Y+X
440 IF Y>800 THEN 480
450 X=0: D=1
460 RETURN
470
480 X=Y-800
490 Y=804: D=3
500 RETURN
```

PERSIAN

This program produces a pattern by drawing hundreds of lines. Random colours are selected by lines 60 and 70. Line 80 moves the origin (middle) of the picture to the middle of the screen.

```
 10 REM PERSIAN
 20 REM ACORN COMPUTERS
 30 REM VERSION 2/16 NOV 81
 40 MODE 1
 50 D%=4
 60 VDU 19,2,RND(3)+1,0,0,0
 70 VDU 19,3,RND(3)+4,0,0,0
 80 VDU 29,640;400;
 90 J1%=0
100 FOR K%=400 TO 280 STEP -40
110 REPEAT J2%=RND(3): UNTIL J2%<>J1%
120 J1%=J2%
130 GCOL 3,J1%
140 FOR I%=-K% TO K% STEP D%
150 MOVE K%,I%
160 DRAW -K%,-I%
170 MOVE I%,-K%
180 DRAW -I%,K%
190 NEXT
200 NEXT
```

SQR ROOT

This program calculates the square root of a number by repeating a simple operation (line 90 and 200) until the calculated result stays steady. The program also indicates how long the calculation takes.

This program illustrates an important mathematical technique but of course you don't have to work out square roots this way – the function **SQR** is provided in **BASIC**.

```
 10 REM ROOT
 20 REM VERSION 1.0 / 16 NOV 81
 30 REM TRADITIONAL ITERATION METHOD
 40 REM TO CALCULATE THE SQUARE ROOT
 50 REM OF A NUMBER TO 3 DECIMAL PLACES
 60 MODE 7
 70 ON ERROR GOTO 300
 80 @%=&2030A
 90 REPEAT
100 count=0
110 REPEAT
120 INPUT "What is your number ",N
130 UNTIL N>0
140 DELTA=N
150 ROOT=N/2
160 T=TIME
170 REPEAT
180 count=count+1
190 DELTA=(N/ROOT-ROOT)/2
200 ROOT=ROOT+DELTA
210 UNTIL ABS(DELTA) <0.001
220 T=TIME-T
230 PRINT
240 PRINT "Number ",N
250 PRINT "Root ",ROOT
260 PRINT "Iterations",count
270 PRINT "Time",T/100;" seconds"
280 PRINT''
290 UNTIL FALSE
300 @%=&90A:PRINT:REPORT:PRINT

>RUN
What is your number?34
Number 34.000
Root 5.831
Interations 5.000
Time 0.070 seconds
What is your number?125
```

```
Number 125.000
Root 11.180
Interations 6.000
Time 0.080 seconds
What is your number?
```

BRIAN

This program prints a 'path in the grass'.

It is a fine example of a 'non-structured' use of **BASIC**; you might like to try and 'structure' it.

```
 90 REM BRIAN2
100 REM(C) BRIAN R SMITH 1980
110 REM ROYAL COLLEGE OF ART, LONDON
120 REM VERSION 1.0 /16 NOV 81
130 INPUT "NUMBER OF CYCLES e.g. 1 to 5 ",T
140 INPUT "BACKGROUND SYMBOL e.g. +",D$
150 INPUT "MOTIF(<20 chrs.)",A$
160 INPUT "TEXT AFTER DESIGN",B$
170 CLS
180 F=1
190 READ A,G,S,C,D,N
200 H=(D-C)/N
210 X=0
220 J=1
230 X=X+S
240 Y=SIN(X)
250 Y1=1+INT((Y-C)/H+0.5)
260 I=0
270 I=I+1
280 IF I=Y1 THEN 310
290 PRINT D$;
300 GOTO 420
310 Z=Z+F
320 IF Z>0 THEN 350
330 F=-F
340 GOTO 450
350 IF Z<=LEN(A$) THEN 390
360 F=-F
370 Z=Z-1
380 GOTO 310
390 S$=LEFT$(A$,Z)
400 PRINT S$;
410 I=I+Z
420 IF I<40 THEN 270
```

```
430 PRINT
440 GOTO 230
450 J=J+1
460 IF J>T THEN 490
470 Z=Z+1
480 GOTO 310
490 FOR K=1 TO 39
500 PRINT D$,
510 NEXT K
520 PRINT
530 PRINT B$
540 DATA 0,6.4,0.2,-1,1,20
>RUN
NUMBER OF CYCLES e.g. 1 to 5 ?3
BACKGROUND SYMBOL e.g. +?.
MOTIF(<20 chrs.)?Hello David!!
TEXT AFTER DESIGN?That's all folks
```

SINE

This program draws a sine wave on the screen. The computer can draw dotted lines and the feature is used to fill in one part of the sine wave (line 130).

The computer can also print letters anywhere on the screen not just on a 40 by 25 grid. Lines 190 to 220 print a message in the shape of another sine curve.

```
 10 REM SINE
 20 REM JOHN A COLL
 30 REM VERSION 2 / 16 NOV 81
 50 MODE 4
 60 VDU5
 70 GCOL 0,1
 80 VDU19,1,1,0,0,0
 90 MOVE 16,400
100
110 FOR X=0 TO 320
120 IF X<150 THEN MOVE 4*X+16,400
130 PLOT 21,4*X+16,300*SIN(X/48)+400
140 NEXT
160 GCOL 0,1
170 A$="SINE WAVES ARE FAR MORE INTERESTING . . . . ."
180
190 FOR X=1 TO 39
200 MOVE X*1280/40,300*SIN(X/6)+500
210 PRINT MID$(A$,X,1)
220 NEXT
230
```

```
240 VDU4
250 END
```

DOUBLE HEIGHT

Here is an example of an assembly language program embedded within a **BASIC** program between the two brackets [and] which enables you to type in double height letters on the screen.

```
 10 REM DOUBLE HEIGHT IN TELETEXT
 20 WIDTH 36: MODE 7
 30 VDU 28,0,23,39,0
 40 write=!&20E AND &FFFF
 50 DIM PROG 100
 60 FOR PASS = 0 TO 1
 70 P%= PROG
 80 [
 90 OPT PASS*3
100 CMP#&D : BNE noter
110 PHA :JSR write
120 LDA#&8D : JSR write
140 LDA#&08 : JSR write
150 LDA#&8D : JSR write
160 PLA : RTS
170 .noter CMP #&20 : BCS legal
180 JMP write
190 .legal PHA : JSR write
200 LDA #$0B : JSR write
210 LDA #&08 : JSR write
220 PLA : PHA : JSR write
230 LDA #&0A : JSR write
240 PLA : RTS
250 ]
260 NEXT PASS
270 !&20E=!&20E AND &FFFF0000 OR PROG
280 END
```

Line 270 changes the 'write character' routine indirection vector so that all output is sent to the new routine given above. This routine tests for a 'return' code (line 100) and if it finds one it issues Teletext double height control codes on to the next two lines. Otherwise the routine just prints the characters on two lines one above the other so as to produce a double height character. This routine has a quite different effect in non-Teletext modes. Try it. Press **<BREAK>** after you have finished with this program.

Before we leave this section, here are a few points about entering lines into **BASIC**.

1. Control characters, for example <CTRL> <B>, will only be 'reflected' in **BASIC** and not entered into any program lines, strings etc.
2. Spaces entered in lines will be preserved, including those at the end of the line. This allows blank lines to be entered e.g.

```
10 <SPACE> <RETURN>
```

to separate program sections. Some of the programs above have such blank lines. Because of this you should avoid using <COPY> past the true end of a line.

3. Most keywords can be abbreviated using a full stop, e.g. `L.` for `LIST`, `SA.` for `SAVE`.

5. BASIC keywords

This chapter contains a detailed description of every word that **BASIC** understands. These words are called 'keywords'.

Some parts of the description are intended for the novice user and others for the person who is familiar with **BASIC**. Each keyword is described under a number of headings as follows:

Keyword

Sometimes followed by a few words explaining the derivation of the word.

Purpose

A plain-English description of what the keyword does. This is intended for the person who is learning **BASIC**.

The only technical terms used are 'string' and 'numeric'. The mathematical functions `SIN, COS, TAN` etc. are not explained for the absolute beginner – there just isn't room to explain everything!

Examples

This section gives a few one-line examples of the keyword (not complete programs). Some of the examples have a number at the start of the line. This number is an 'example line number'.

The examples are only intended to be illustrative. In some cases a line of **BASIC** program may overflow onto the next line as elsewhere in this book.

Description

In this section the keyword is described using normal computer jargon.

Syntax

The syntax of each keyword's usage is given more by way of helpful explanation than for its strict accuracy. Purists will, rightly, complain at travesty of Backus-Naur form. Others may find the entries useful.

The following symbols are used as part of the syntax explanation:

{ }	Denote possible repetition of the enclosed symbols zero or more times.
[ ]	Enclose optional items.
|	Indicates alternatives from which one should be chosen.
<num-const>	Means a numeric constant such as '4.5' or '127'
<num-var>	Means a numeric variable such as 'X' or 'length'
<numeric>	Means either a <num-const> or a <num-var>, or a combination of these in an expression such as "4*X+6"
<string-const>	Means a string enclosed in quotation marks, e.g. "ROBERT BULL".
<string-var>	Means a string variable such as **A$** or **NAME$.**
<string>	Means either a <string-const> or a <string-var>, or an expression such as **A$+"LINDA"**.
<testable condition>	Means something which is either **TRUE** or **FALSE**. Since **TRUE** and **FALSE** have values it is possible to use a <numeric> at any point where a <testable condition> is required. The distinction between these two is, in fact, rather unnecessary.
<statement>	Means any **BASIC** statement, for example, **PRINT** or **GOSUB** or **PROC**.
<variable name>	Means any sequence of letters or numbers that obeys the rules for variables.

Associated keywords

This section is intended to draw your attention to other keywords which either have similar functions or which are normally used in conjunction with this keyword. You will probably find it helpful to read the pages which describe the associated keywords.

Demonstration program

If appropriate a short program is included to illustrate the use of the keyword. Parentheses are generally optional where sense is unaffected.

ABS absolute value

Purpose

This function turns negative numbers into equivalent positive numbers but leaves positive numbers alone. For example the absolute value of -9.75 is 9.75 while the absolute value of 4.56 is 4.56.

The **ABS** function is often used when calculating the difference between two values if you do not know which is the larger of the two. Thus (K-L) will be positive if K is greater than L, and will be negative if L is greater than K.

For example, if K = 9 and L = 12 then (K - L) would be equal to -3. However the value of **ABS(K-L)** will always be positive. In the example given **ABS(K-L)** would equal 3.

Examples

```
205 error=ABS(DIFFERENCE)
100 DIFF=ABS(X2-X1)
PRINT ABS(temp%-50)
```

Description

A function giving the absolute value of its argument.

Syntax

<num-var>=**ABS**(<numeric>)

Associated keywords

SGN

ACS arc-cosine

Purpose

To calculate an angle whose cosine is known. The calculated angle is in radians but may be converted to degrees by using the function **DEG**. See **DEG** for more information.

Examples

```
10 X=ACS(Y)
1205 angle=DEG(ACS(0.5678))
330 OUT=ACS(.234)
PRINT ACS (0.5)
```

Description

A function giving the arc-cosine of its argument. The result is in radian measure.

Syntax

<num-var>=**ACS**(<numeric>)

Associated keywords

ASN, ATN, SIN, COS, TAN, RAD, DEG

ADVAL analogue to digital converter value

Purpose

An analogue signal is one which can have almost any value – including fractional parts. It is contrasted with a digital signal which is expressed in exact numbers. The height of the water in a harbour is an analogue quantity whereas the number of boats it contains is a digital quantity.

Watches always used to have analogue dials – 'The time is about four fifteen'. Electronic things usually work with whole numbers; for example

16h : 15m : 23s

There are four 'analogue to digital' converters in the BBC Microcomputer. Each analogue to digital converter in the computer accepts a voltage and gives out a whole number indicating how large the voltage is. This voltage might be controlled by, for example, the position of a 'games paddle' or 'joystick' control which is connected to the computer. Alternatively the computer might be connected to a speed sensor on a piece of machinery or it might measure the temperature of a room.

The input voltage range is 0 volt to 1.8 volts. When the input is 0V the converter produces the number zero. With 1.8V input the converter produces the number 65520. Why 65520? The circuit in the computer which does the conversion was designed to give out numbers in the range 0 to 4095. However it may well be that future converters can give out numbers over a larger range – enabling the computer to measure things more accurately. In order to ensure that the BBC Microcomputer can be used in this situation, a large range has been specified.

Instead of producing numbers in the 0 to 4095 range it produces a number in the range 0 to 65520. Therefore instead of numeric results going up in the sequence 0, 1, 2, 3 etc they will go 0, 16, 32, 48, 64 etc. If you prefer the range 0 to 4095 then just divide the value by 16.

There are four analogue input channels provided in the BBC Microcomputer and the number in parentheses after the keyword **ADVAL** refers to the channel whose value you wish to find. The channels are numbered 1, 2, 3 and 4.

ADVAL(0) performs a special function in that it can be used to test to see which of the 'fire' buttons is pressed on the games paddles. The value returned also indicates which ADC channel was the last one to be updated. The following can be used to extract these two pieces of information from the value returned by **ADVAL(0)**:

```
X=ADVAL(0) AND 3
```

will give a number with the following meaning:

```
X=0 no button pressed
X=1 left side fire button pressed
X=2 right side fire button pressed
X=3 both fire buttons pressed
```

```
X=ADVAL(0) DIV 256
```

will give the number of the last analogue to digital channel to complete conversion. If the value returned is zero then no channel has yet completed conversion.

ADVAL with a negative number in the parentheses, e.g. **X=ADVAL(-3)**, can be used to see how full any of the internal buffers are. When characters are typed in on the keyboard they are put into a buffer from which they are extracted with statements like **INPUT** and **GET**. Other buffers are used internally for other purposes. The exact meaning of the number returned depends on the buffer being tested.

X=ADVAL(-1) Returns the number of characters in the keyboard buffer.

X=ADVAL(-2) Returns the number of characters in the RS423 input buffer.

X=ADVAL(-3) Returns the number of free spaces in the RS423 output buffer.

X=ADVAL(-4) Returns the number of free spaces in the printer output buffer.

X=ADVAL(-5) Returns the number of free spaces in the sound channel 0 buffer.

`X=ADVAL(-6)` Returns the number of free spaces in the sound channel 1 buffer.

`X=ADVAL(-7)` Returns the number of free spaces in the sound channel 2 buffer.

`X=ADVAL(-8)` Returns the number of free spaces in the sound channel 3 buffer.

`X=ADVAL(-9)` Returns the number of free spaces in the speech buffer.

This feature can be used, for example, to ensure that a program never gets stuck waiting for a sound channel to empty, e.g.:

```
IF ADVAL(-7)<> 0 THEN SOUND 2,...etc
```

Examples

```
980 X=ADVAL(3)
125 TEMP=ADVAL(X)
intensity=ADVAL(1)
```

Syntax

<num-var>=**ADVAL**(<numeric>)

Description

A function which returns the last known value of the analogue to digital channel given in its argument. There are four channels, each of 10 bit resolution, but the returned value is scaled to 16 bits.

The analogue to digital converter cycles repeatedly through the selected channels and keeps a table of the result so that the function **ADVAL** returns very quickly. New samples are taken about every ten milliseconds. Therefore with four channels selected results will be updated every 40ms.

AND

Purpose

`AND` can be used either as a logical operator or as a 'bit by bit', or 'Boolean' operator.

As a logical operator `AND` is used to ensure that two conditions are met before something is done. For example:

```
IF X=9 AND Y=0 THEN PRINT "HELLO"
```

Logical `AND` is most often used as part of an `IF...AND...THEN...` construction.

Boolean `AND` compares the first bit of one number with the first bit of another number. If both bits are set to a one (rather than a zero) then the first bit in the answer is also set to a one. This process is then repeated for the second bit in each of the two numbers being compared and so on for all 32 bits in the numbers. For example the result of `14 AND 7` is `6`, since in binary:

14	**is**	**0000 0000 0000 0000 0000 0000 0000 1110**
7	**is**	**0000 0000 0000 0000 0000 0000 0000 0111**
6	**is**	**0000 0000 0000 0000 0000 0000 0000 0110**

Examples

```
300 IF length>9 AND wt>9 THEN PRINT "YES"
100 IF X=2 AND cost>5 AND J=12 THEN PRINT "NO!!"
```

The above example will only print `NO!!` if all three conditions are met.

Description

The operation of Boolean **AND** between two items. Note that the logical and Boolean operations are in fact equivalent. This follows since the value of **TRUE** is -1 which is represented on this machine by the binary number:

1111 1111 1111 1111 1111 1111 1111 1111

Similarly the binary value of **FALSE** is

0000 0000 0000 0000 0000 0000 0000 0000

Thus **PRINT 6=6** would print **-1** since **6=6** is **TRUE**.

Syntax

<num-var> = <numeric> **AND** <numeric>

or

<testable condition> = <testable condition> **AND** <testable condition>

Associated keywords

EOR, OR, FALSE, TRUE, NOT

ASC American Standard Code (ASCII)

Purpose

There are two commonly used methods of talking about characters (things like A, B, 5, ?, and so on). Obviously they are single characters! So we can say `D$="H"` - meaning put the letter H into the box in the computer labelled `D$`. The computer understands this but it doesn't actually put an H into the box, Instead it stores a number which represents the letter H (in fact the number is 72). Every character has a unique corresponding number called its ASCII code. (ASCII stands for American Standard Code for Information Interchange. The abbreviation ASCII rhymes with 'Laski'.)

Sometimes it is convenient to find out what number corresponds to a particular character - that is its ASCII code. You can look it up at Appendix C, or you can say to the computer:

```
PRINT ASC("H")
```

The function `ASC` gives the ASCII value of the first letter in the string. Thus:

```
PRINT ASC("Good")
```

gives 71, the ASCII value of 'G'.

The reverse process of generating a one-character string from a given ASCII value is performed by the function `CHR$`.

Description

A function returning the ASCII character value of the first character of the argument string. If the string is null (empty) then -1 will be returned.

Syntax

<num-var> = `ASC`(<string>)

Associated keywords

CHR$, STR$, VAL

ASN arc-sine

Purpose

To calculate an angle whose sine is known. The calculated angle is in radians but may be converted to degrees by using the function **DEG**. A radian is equal to about 57 degrees. Mathematicians often prefer to work in radians.

Examples

```
340 J=ASN(0.3456)
30 angle=DEG(ASN(.7654))
PRINT ASN(.5)
```

Description

A function giving the arc-sine of its argument. The result is in radian measure.

Syntax

<num-var>=**ASN**(<numeric>)

Associated keywords

ACS, ATN, SIN, COS, TAN, RAD, DEG

ATN arc-tangent

Purpose

To calculate an angle whose tangent is known. The calculated angle is in radians but may be converted to degrees by using the function **DEG**.

Examples

```
1250 X=ATN(Y)
240 value=DEG(ATN(22.31))
```

Description

A function giving the arc-tangent of its argument. The result is in radian measure.

Syntax

<num-var>=**ATN**(<numeric>)

Associated keywords

ACS, ASN, SIN, COS, TAN, RAD, DEG

AUTO automatic

Purpose

When typing a **BASIC** program into the computer it is common to make the first line of the program line number 10, the second line 20 etc. To save having to type in the line number each time, the command `AUTO` can be used to make the computer 'offer' each line number automatically. Used on its own the command `AUTO` will offer first line 10 and then lines 20, 30, 40 etc. The command `AUTO 455` would instead start the process at line number 455, followed by lines 465, 475, 485 etc.

Another option allows the user to select the step size. Thus the command `AUTO 465,2` would cause the computer to offer lines 465, 467, 469, 471 etc. The largest step size is 255.

To escape from `AUTO` mode the user must press the **<ESCAPE>** key. `AUTO` mode will be abandoned if the computer tries to generate a line number greater than 32767.

Examples

```
AUTO
AUTO 220
AUTO 105,5
```

Syntax

`AUTO` [<num-const>[`,`<num-const>]]

Description

`AUTO` is a command allowing the user to enter lines without first typing in the number of the line. Because `AUTO` is a command it cannot form part of a multiple statement line. `AUTO` mode can be left by pressing **<ESCAPE>** or generating a line number exceeding 32767. `AUTO` may have up to two arguments. The first optional argument gives the starting line number and the second optional argument gives the increment between line numbers.

BGET# get a byte from file

Purpose

Numbers and words can be recorded on cassette tape, disc, Compact Flash hard drives and other storage mediums. The function **BGET#** enables a single character or number to be read back into the computer from the storage device. Before using this statement a file must have been opened using the **OPENIN** function or else an error will occur. When a file is opened, using **OPENIN**, the computer will allocate the file a channel number. This number must be used in all subsequent operations on the file, for example when reading the file or when writing a new file. See the chapter on file handling in the manual for the filing system you are using.

Examples

```
6000 character=BGET# (channel)
340 next_letter%=BGET#C
```

Description

A function which gets a byte from the file whose channel number is the argument. The file must have been opened before this statement is executed.

Syntax

<num-var>=**BGET#**<num-var>

Associated keywords

OPENIN, OPENUP, OPENOUT, CLOSE#, EXT#, PTR#, PRINT#, INPUT#, BPUT#, EOF#

BPUT# put a byte to file

Purpose

To store a byte on cassette, disc or other storage medium. The number which is sent to the file can have any value between 0 and 255. If you attempt to store a number that is greater than 255, then 256 will be repeatedly subtracted from the number until it is less than 256. The final number will then be sent to file. (This statement is used to store single bytes – not large numbers. Larger numbers can be stored using **PRINT#**.) As with **BGET#** the file must be 'open' before this statement can be used. See the chapter on file handling in the manual for the filing system you are using.

Examples

```
30 BPUT# channel,number
700 BPUT#N,32
450 BPUT# STAFF_FILE, A/256
```

Description

A statement which puts a byte to the file whose channel number is the first argument. The second argument's least significant byte is sent. The file must be open before this statement is executed.

Syntax

BPUT#<num-var>,<numeric>

Associated keywords

OPENIN, OPENUP, OPENOUT, CLOSE#, EXT#, PTR#, PRINT#, INPUT#, BGET#, EOF#

CALL transfer control to a machine code subroutine

Purpose

This statement makes the computer execute a piece of machine code which the user has previously placed in the computer's memory. Before using this powerful statement you should have a good understanding of machine code and assembly language as incorrect use can destroy a program completely! Programming in assembler and machine code is covered in Chapters 11 and 12.

Examples

```
50 rotate=&0270
60 CALL rotate,J,K,L
200 CALL 1234,A$,M,J$
```

Description

A statement to call a piece of machine code. The number of parameters passed is variable and may be zero. The parameters are variable parameters and may be changed on execution of the subroutine. The addresses of parameters are passed in a parameter block starting at location &0600 hex.

On entry to the subroutine the processor's A, X and Y registers are initialised to the least significant bytes of the integer variables A%, X% and Y%. The carry flag is set to the least significant bit of the C% variable.

On entry a parameter block is set up by the computer and it contains the following list:

Number of parameters	1 byte
Parameter address	2 bytes
Parameter type	1 byte
Parameter address	Repeated as often
Parameter type	as necessary

Parameter types

0 - 8 bit byte (e.g. **?X**)

4 - 32 bit integer variable (e.g. **!X** or **X%**)

5 - 40 bit floating point number (e.g. **V**)

128 - A string at a defined address (e.g. **$X** - terminated by a &0D)

129 - A string variable such as **A$**

In the case of a string variable the parameter address is the address of a string information block which gives the start address, number of bytes allocated and current length of the string in that order.

Syntax

CALL<numeric>{ , <num-var> | <string-var>}

Associated keywords

USR

CHAIN

Purpose

CHAIN enables a program to be split up into a number of small sections. The **CHAIN** statement is used to enable one program to **LOAD** and **RUN** another program automatically. For example, one program might enable the user to enter the number of hours worked by employees and that program might **CHAIN** a second program which would print out the payslip. In turn that might **CHAIN** a third program which would do a cost per hour analysis on the data held on the file.

CHAIN is also useful in a game with a lot of instructions. The instructions could all be stored as one file which would then **CHAIN** the main game – thus releasing a lot of the computer's memory.

CHAIN"" (without the program name) will chain the next program on a cassette, whatever its file name. This will not work with disc filing systems where you must give the file name. For that reason it must not be used in programs which may be used on disc systems. It will however also work with the ROM filing system, when used it will load and run the first program in the RFS list. See Chapter 20.

Examples

```
900 CHAIN "GAME_1"
1234 CHAIN "NEWPROG"
CHAIN A$
```

Description

A statement which will load and run the program whose name is specified in the argument. All variables except @% and A% to Z% are cleared.

Syntax

CHAIN<string>

Associated keywords

LOAD, SAVE

CHR$ character string

Purpose

To generate a character (single letter or number etc) from the number given in string form. The character generated will be the ASCII character at the position given in the ASCII table. See the description of **ASC** and the full ASCII table in Appendix C.

The statement **VDU** has a similar effect to **PRINT CHR$** and may be more useful in some applications.

Examples

```
220 RED$=CHR$(129)
```

```
1070 PRINT CHR$(8);
```

makes the cursor move left one position.

```
PRINT CHR$(7)
```

causes a short note to be emitted by the loudspeaker.

Description

A string function whose value is a single character string containing the ASCII character specified by the significant byte of the numeric argument. Thus **CHR$(-1)** would give ASCII character number 255.

Note that the statement **VDU** is probably more useful when sending characters to the screen, since it involves less typing. **CHR$** is needed when you wish to put a special character into a string.

Syntax

<string-var>=**CHR$**(<numeric>)

Associated keywords

ASC, STR$, VAL, VDU

CLEAR

Purpose

This tells the computer to forget all variables previously in use, including string variables and arrays but excluding the 'resident integer variables' @% and A% to Z% which are not affected in any way.

Examples

```
350 CLEAR
CLEAR
```

Description

A statement which deletes all variables except the resident integer numeric variables @% and A% to Z%.

Syntax

CLEAR

Associated keywords

None

CLG clear the graphics screen

Purpose

To clear the graphics area of the screen. The graphics area of the screen is left in the colour selected as the 'current graphics background colour'. See the keyword GCOL for more information. The graphics cursor is then moved to its home position (0,0) which is at the bottom left of the graphics area.

Examples

```
870 CLG
CLG
```

Description

Clears the current graphics area of the screen and sets this area to the current graphics background colour in addition. The statement then moves the graphics cursor to the graphics origin (0,0).

Syntax

```
CLG
```

Associated keywords

CLS, GCOL

CLOSE#

Purpose

To inform the computer that you have completely finished with a particular file. The computer then transfers any data still in memory to cassette, disc, hard drive or Econet etc. as needed. See the chapter on file handling in the manual for the filing system you are using.

Example

```
90 CLOSE#N
```

Description

A statement used to close a specific disc or cassette file. `CLOSE#0` will close all files.

Syntax

`CLOSE#`<numeric>

Associated keywords

OPENIN, OPENUP, OPENOUT, EXT#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, EOF#

CLS clear the text screen

Purpose

To clear the text area of the screen. Any graphics in this area will also be cleared. The text area will be left in the 'current text background colour'. The text cursor will then be moved to its 'home' position at the top left of the text area. See the keyword **COLOUR** for more information about text background colours.

Examples

```
560 CLS
CLS
```

Description

Clears the current text area and sets this area of the screen to the current text background colour. The statement then causes the text cursor to move to the text origin (0,0) at the top of the current text area. **CLS** resets **COUNT**.

Syntax

```
CLS
```

Associated keywords

CLG, COLOUR

COLOUR

Purpose

This statement selects the colour in which the computer is to print the text and also its background. The command has a number of variations which are most easily explained by example.

Type in the following:

```
MODE 5
COLOUR 1
COLOUR 2
COLOUR 3
```

and press <RETURN> at the end of each line as usual.

As you will have seen, these commands change the colour of the text. This is often called the 'text foreground colour'. Now try:

```
COLOUR 129
COLOUR 130
COLOUR 128
```

These numbers change the 'text background colour'.

In any two colour `MODE` (`MODE 0, 3, 4` or `6`) the following normally apply:

Foreground	Background	Colour
0	128	Black
1	129	White

In any four colour MODE (MODE 1 or 5) the following normally apply:

Foreground	Background	Colour
0	128	Black
1	129	Red
2	130	Yellow
3	131	White

In MODE 2 the following normally apply:

Foreground	Background	Colour
0	128	Black
1	129	Red
2	130	Green
3	131	Yellow
4	132	Blue
5	133	Magenta (blue/red)
6	134	Cyan (blue/green)
7	135	White (normal foreground)
8	136	Flashing black/white
9	137	Flashing red/cyan
10	138	Flashing green/magenta
11	139	Flashing yellow/blue
12	140	Flashing blue/yellow
13	141	Flashing magenta/green
14	142	Flashing cyan/red
15	143	Flashing white/black

If you are not familiar with **BASIC** then you may already have had too much of this! Nevertheless, it is possible, for example in a four colour MODE to select any four colours from the available 16 effects by using another command. Remember that the colours given above (black, red, yellow, white) will be available as soon as four colour MODE is selected – but you can then select other colours later.

Try the following:

```
MODE 5
COLOUR 1
VDU 19,1,4,0,0,0
VDU 19,1,5,0,0,0
COLOUR 2
VDU 19,2,4,0,0,0
VDU 19,1,3,0,0,0
```

As you will see the statement `VDU 19,` can be used to change the 'actual colour' of colour 1 or 2.

The number which follows the `VDU 19,` is the number that is referred to by the `COLOUR` statement. It is referred to as a 'logical colour'.

The number which follows the 'logical colour' is referred to as the 'actual colour' and is as follows:

0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta (blue/red)
6	Cyan (blue/green)
7	White (normal foreground)
8	Flashing black/white
9	Flashing red/cyan
10	Flashing green/magenta
11	Flashing yellow/blue
12	Flashing blue/yellow
13	Flashing magenta/green
14	Flashing cyan/red
15	Flashing white/black

Thus the statement `VDU 19,3,6,0,0,0` will set logical colour 3 to be cyan. So if in `MODE 4`, a two colour `MODE`, you wanted black letters on a yellow background you would issue the command:

```
VDU 19,1,0,0,0,0
VDU 19,0,3,0,0,0
```

Alternatively, you could string the whole lot together as:

```
VDU 19,1,0,0,0,0,19,0,3,0,0,0
```

This combination of the `COLOUR` statement and the `VDU 19` statement enables a very wide range of effects to be obtained. There are also `*FX` calls which enable the flash rates of the colours to be altered as well, see Chapter 14.

Syntax

`COLOUR`<numeric>

Associated keywords

VDU, GCOL

COS cosine

Purpose

To calculate the cosine of an angle. Note that the number in parenthesis (the angle) is expressed in radians and not in degrees. To convert from degrees to radians use the function RAD.

Examples

```
PRINT COS(2.45)
780 X=COS(Y)
655 Number=COS(RAD(45))
```

Description

A function giving the cosine of its argument. The argument must be given in radians.

Syntax

<num-var>=COS(<numeric>)

Associated keywords

SIN, TAN, ACS, ASN, ATN, DEG, RAD

COUNT

Purpose

COUNT counts all the characters printed using PRINT, whether to screen, printer or RS423 output channel. On the other hand POS returns the current position of the actual text cursor on the screen.

Examples

```
290 A=COUNT
75 fred=COUNT
PRINT COUNT
```

Description

A function returning the number of characters printed since the last new line. COUNT is set to zero if the output stream is changed.

Syntax

<num-var>=COUNT

Associated keywords

POS

Demonstration program

```
 5 REM to print a row of 16 * signs
 7 REM this is not the easiest way!
10 X=16
20 REPEAT PRINT "*";
30 UNTIL COUNT=X
```

DATA

Purpose

DATA is used in conjunction with the keyword **READ**, and sometimes with **RESTORE**, to enable you to make available automatically any data (numbers and words) that will be needed by a program.

For example, if you were writing a geography quiz, you might want to use the names of five countries and their five capital cities each time you used the program. The names of the cities and countries can be entered as **DATA** in the program and will always be there when the program is run.

Computers using the language **BASIC** are really clumsy at handling information like this, as the demonstration program on the next page shows.

In the example program the **DATA** consists of many words. **DATA** statements can just as well contain numbers – or a mixture of words and numbers. In our example the words were all read into a string array.

It is essential that the **DATA** contains numbers where numeric variables are to be filled. Text information, e.g. **hello**, will just give 0.

There is no need to put each word in quotes unless leading spaces are important in the **DATA** words, for example **" four spaces".**

If you wish to have leading spaces then these words should be enclosed in quotes. Since a comma is used to separate items of **DATA**, if you want a comma in your **DATA**, you must enclose the **DATA** in quotes.

Examples

```
100 DATA "Allen, Stephen",Stamp dealer,01-246,8007,24
130 DATA "TOP OF ROOF", 450, January
```

Description

A program object which must precede all lists of data intended for use by the **READ** statement.

Syntax

DATA<str-const>|<num-const>{,str-const>|<num-const>}

Associated keywords

READ, RESTORE

Demonstration program

```
 10 REM geography quiz
 20 DIM city$(5)
 30 DIM country$(5)
 40 FOR x=1 TO 5
 50 READ city$(x)
 60 READ country$(x)
 70 NEXT x
 80 right=0
110 FOR x=1 TO 10
120 r=RND(5)
130 PRINT "What city is the capital"
140 PRINT "of "; country$(r)
150 INPUT answer$
160 IF answer$=city$(r) THEN PROCyes ELSE PROCno
170 NEXT x
180 PRINT "You got ";right;
190 PRINT " correct out of 10"
200 END
500 DATA Paris, France, Reykjavik
505 DATA Iceland
510 DATA Moscow, Soviet Union
520 DATA Athens, Greece
530 DATA Spitzbergen, Spitzbergen
600 DEF PROCyes
610 PRINT "Well done!"
620 right=right+1
630 ENDPROC
700 DEF PROCno
710 PRINT "No, the capital of
720 PRINT country$(r);" is ";city$(r)
730 ENDPROC
```

Line 10 is just a **REM**ark which the computer ignores.

Lines 20 and 30 tell the computer that we are going to use two string arrays – one to store the names of the five cities and the other to store the names of the five countries.

Line 40 sets up a **FOR...NEXT** loop that will go around five times.

Line 50 reads the next word (which will be a city) into the array **city$** and then moves the 'data pointer' on to point to the next word (which will be a country).

Line 60 reads the next piece of **DATA** into the **country$** array.

Line 70 is the end marker of the **FOR...NEXT** loop.

Lines 110 to 170 loop ten times through a 'question and answer' quiz.

Lines 500 to 530 contain the **DATA** used above.

Lines 600 to 630 are a procedure to deal with correct replies.

Lines 700 to 730 deal with incorrect replies.

DEF define

Purpose

The word **DEF** is used to inform the computer that a procedure or function is about to be defined. Once the computer has been informed that this procedure or function exists, then the procedure or function can be called by name anywhere in the program.

Definitions of procedures and functions ***must not occur in the body of a program***. They should be placed in a separate section ***which is not executed*** – for example after the final **END** in the program. This also aids readability.

The language **BASIC** has many predefined functions which the computer already knows about. For example, the function **SQR** enables it to calculate the square root of a number.

Often though, it is useful to be able to define your own functions. For example, you might want to have a function which calculates the VAT inclusive price of a product from the **BASIC** sale price by multiplying by 1.175.

A function always produces a result so you can write **X=FNST**. A procedure, on the other hand, is used to perform a number of actions, but it does not by itself produce a numerical result. For example, a procedure might be set up to clear the screen and draw a number of lines on the screen.

You may well feel confused, but do not be put off! The use of procedures and functions may be difficult to understand at first but it is well worth the effort. Their use greatly enhances the readability and reliability of programs.

The section below gives a more detailed explanation of the use of procedures and functions. It should be read in conjunction with the examples which follow.

Both procedures and functions may contain local variables which are declared using the word **LOCAL**. In the third example given below, K is declared as a local variable. This means that although K is used in this procedure its value is not defined when the procedure finishes. In fact the variable K might well be used elsewhere in the program. The variable K, elsewhere within the program, would

not be altered by the use of the local variable K within the procedure. Any variable which is not declared as **LOCAL** will be available outside the procedure, in other words to the rest of the program.

Also, both procedures and functions may have parameters passed to them. Look at the first example program below. Line 1010 says:

```
1010 DEF FNST(g)=1.15*g
```

'g' is called a 'formal parameter' for the function **FNST**. It tells the computer that one number is going to be passed to the function when the function is used and inside the function we have decided to use the letter g to represent the variable.

The procedure is 'called' or used like this – for example:

```
230 PRINT "VAT inclusive price ";
236 PRINT FNST(P)
```

and in this case 'P' is the 'actual parameter' for the function **FNST**. Whatever value 'P' has will be used inside the function wherever reference is made to the formal parameter 'g'. This is very convenient since you can use any variable names you like for the parameters inside the procedure. Then you can call the procedure with a quite different set of parameter names from the outside. Very often a procedure will be called from many different places in the program – and the actual parameters may have different names each time the procedure or function is called.

If a procedure or function is defined with (say) three formal parameters then, when it is called, three actual parameters must be supplied. See the fifth example below where three parameters are passed to the function.

The end of the procedure is indicated with the statement **ENDPROC**. The end of a multi-line function is indicated by the statement that starts with an = sign. The function is given the value of the expression to the right of the = sign.

Examples

First example – full program:

```
 210 REPEAT
 220 INPUT "BASIC price ",P
 230 PRINT "VAT inclusive price ";
 235 PRINT FNST(P)
 240 UNTIL P=0
 250 END
1000 REM line numeric function
1010 DEF FNST(g)=1.15*g
```

Second example – program section

Multi-line string function with one string parameter:

```
1000 DEF FNREVERSE(A$)
1010 REM reverse the order of the letters in A$
1015 REM
1020 LOCAL d%,B$
1030 FOR d%=1 TO LEN(A$)
1040 B$=MID$(A$,d%,1)+B$
1050 NEXT d%
1060 =B$
```

Third example – program section

Multi-line procedure with one parameter:

```
 200 DEF PROCbye(X)
 210 REM print bye X times
 220 LOCAL K
 230 FOR K=1 TO X
 240 PRINT "bye"
 250 NEXT K
 260 ENDPROC
```

Fourth example - program section

This sets the background colour to a new value given in the parameter:

```
10 DEF PROCINITSCREEN(X)
20 REM clear screen and draw border
25 COLOR 128+X
30 CLS
40 DRAW 1279,0
50 DRAW 1279,799
60 DRAW 0,799
70 DRAW 0,0
80 ENDPROC
```

Fifth example - full program:

```
 110 INPUT X,Y,Z
 120 M=FNMEAN(X,Y,Z)
 130 PRINT "The mean of ",X,Y,Z
 140 PRINT "is ";M
 150 END
8990 REM Single line numeric function
8995 REM with three parameters
9000 DEF FNMEAN(A,B,C)=(A+B+C)/3
```

Description

A program object which must precede declaration of a user function or procedure. String and numeric functions and procedures may be defined. Multi-line functions and procedures are allowed. All procedures and functions must be placed in the program where they will not be executed, e.g. after the **END** statement.

Syntax

DEF FN|**PROC**<variable name>[(<string>|<numeric>{,<string>|<numeric>})]

Associated keywords

ENDPROC, FN, PROC

DEG degrees

Purpose

This function converts angles which are expressed in radians into degrees. A radian is equal to about 57 degrees.

Examples

```
100 X=DEG(PI/2)
300 angle=DEG (1.36)
PRINT DEG(PI/2)
```

Syntax

<num-var>=**DEG**<numeric>

Description

A function which converts radians to degrees.

Associated keywords

RAD, SIN, COS, TAN, ACS, ASN, ATN

DELETE

Purpose

The **DELETE** command is used to delete a group of lines from a program. It cannot be used as part of a program. You can specify which lines should be deleted with a command of the form:

```
DELETE 120,340
```

This would remove everything between line 120 and line 340 inclusive. To delete everything up to a certain line number (for example up to line 290) use **DELETE 0,290.**

To delete from line 500 to the last line, use as the last line to be deleted any number greater than the last line number in the program. Since the largest line number allowed is 32767:

```
DELETE 500,32767
```

will do the trick, but will take a long time. To delete a single line just type the line number and press **<RETURN>**. There is no need to use the **DELETE** command.

Examples

```
DELETE 0,540
DELETE 180,753
```

Syntax

DELETE<num-const>,<num-const>

Description

A command enabling a range of lines to be deleted from a program. Since **DELETE** is a command it cannot be used in a program or as part of a multiple statement line.

Associated keywords

LIST, OLD, NEW

DIM dimension of an array

Purpose

As well as simple numeric and string variables (such as 'X' and 'name$') it is possible to work with 'arrays' of variables. These are extremely useful when working with groups of numbers or words. For example if you wanted to work with a set of information about the rooms in a hotel with four floors, each with 30 rooms, then an array of 4 by 30 entries can be created like this:

```
DIM hotel(4,30)
```

Having set up an array, one can enter information into each of its 'elements'. For example the cost of the room per night might be £26.50:

```
hotel(1,22)=26.50
hotel(4,1)=165.00
```

In practice the statement `DIM hotel(4,30)` produces an array of five by 31 entries since the lowest array element is `hotel(0,0)`.

All the above arrays are called 'two dimension numeric arrays'. Another array could contain the names of guests:

```
DIM name$(4,30)
name$(1,22)="Fred Smith"
name$(4,1)="The Queen"
```

That sort of array is called a 'two dimension string array'.

Arrays may have one or more dimensions. A single dimension array would be appropriate for all the houses in a road, e.g.:

```
DIM MainSt(150)
```

That sort of array is called a 'single dimension numeric array'. All arrays are normally dimensioned very early in the program. It is 'illegal' to attempt to change the size of an array by re-dimensioning it later in the program. An array may have

as many dimensions and as many elements in each dimension as the computer has space for – but you tend to run out of computer memory pretty fast with large arrays! It is essential that there is no space between the array name and the first parenthesis. Thus `DIM A(10)` is correct but `DIM A (10)` will not define an array.

Examples

```
100 DIM partnumbers(1000)
3000 DIM employeename$(35)
240 DIM ALL_hours_in_the_week(24,7)
100 DIM A(X)
```

Description

A statement which dimensions arrays. Arrays must be predeclared before use. After dimensioning all elements of arrays are initialised to zero for numeric arrays or null strings in the case of string arrays. The lowest element in an array is element zero. Thus `DIMX(4)` would create an array of five elements (0 to 4 inclusive).

There is a second and quite different use for the `DIM` statement. It can be used to reserve bytes in memory for special applications. To reserve 25 bytes, type `DIM X 24`.

Notice two things about this statement: firstly the space between the variable `X` and the (number of bytes minus 1) and secondly the absence of parentheses around the `24`. The address of the start of the group of 25 bytes is given in the variable `X` in this example.

Syntax

`DIM`<num-var>|<str-var>(<numeric>{,<numeric>})

or

`DIM`<num-var> <numeric>

Associated keywords

None

DIV division of whole numbers

Purpose

See the keyword **MOD** for a full explanation. **DIV** is an operator which gives the whole number part of the result of a division. Thus:

```
PRINT 11 DIV 4
```

gives 2 (leaving a remainder of 3).

Description

A binary operator performing integer division between its operands. The operands are truncated to integers before division takes place.

Syntax

<num-var>=<numeric>**DIV**<numeric>

Associated keywords

MOD

DRAW

Purpose

This statement draws lines on the screen in MODEs 0, 1, 2, 4 and 5. The DRAW statement is followed by two numbers which are the X and Y coordinates of the end of the line. The line starting point can either be the end of the last line that was drawn or else a new point if the MOVE statement has been used before the statement DRAW.

The screen is addressed as:

1280 points wide X-axis, 0-1279
1024 points high Y-axis, 0-1023

regardless of the graphics MODE selected. The origin (position 0,0) is normally at the bottom left of the screen.

The line is drawn in the current graphics foreground colour. This can be changed using the GCOL statement.

Examples

```
780 DRAW X,Y
DRAW 135,200
```

Description

DRAW X,Y means draw a line to X,Y in the current foreground colour.

DRAW X,Y is equivalent to PLOT 5,X,Y.

DRAW is one of a large group of line drawing statements. See PLOT for others.

Syntax

DRAW<numeric>,<numeric>

Associated keywords

MODE, PLOT, MOVE, CLG, VDU, GCOL

Demonstration program

```
140 MODE 5
160 REM Red background
170 GCOL 0,129
175 CLG
180 REM Yellow foreground
190 GCOL 0,2
200 REM draw a box
210 MOVE 100,100
220 DRAW 400,100
230 DRAW 400,400
240 DRAW 100,400
250 DRAW 100,100
```

ELSE

Purpose

To provide an alternative course of action. **ELSE** can be used following an **IF...THEN** statement, an **ON...GOTO** statement, and an **ON...GOSUB** statement.

Examples

```
560 IF length > 0 THEN PRINT "OK" ELSE PRINT "No good"
100 IF A<>B THEN C=D ELSE PRINT "Values match"
```

Description

Part of the **IF...THEN...ELSE** structure.

Syntax

IF<testable condition>[**THEN**]<statement>[**ELSE**<statement>]

or

ON<num-var>**GOTO**<numeric>{,<numeric>}[**ELSE**<statement>]

or

ON<num-var>**GOSUB**<numeric>{,<numeric>}[**ELSE**<statement>]

Associated keywords

IF, THEN, ON

END

Purpose

This informs the computer that it has reached the end of the program. **END** is optional but may be used as many times as required in a program.

Example

```
9000 END
```

Description

Optional end of program which may occur anywhere and as often as is required. The command **END** has a special use in that it causes **BASIC** to search the program in memory for a valid end program marker. **BASIC** then updates its internal pointers. This may be useful after unusual loading procedures. If the user changes the value of **PAGE** then internal pointers such as **TOP** will not be reset until an **END** statement or command is met.

Syntax

END

Associated keywords

STOP

ENDPROC end procedure

Purpose

This indicates the end of a procedure definition. See the keyword **DEF** for more information.

Examples

```
1000 DEF PROCdash(param)
1010 REM print dashes lots of times
1020 REM in fact "param" dashes in total
1025 REM
1030 LOCAL counter
1040 FOR counter=1 TO param
1050 PRINT"-";
1060 NEXT counter
1070 ENDPROC
2010 DEF PROCtriangle(A,B,C,D,E,F)
2020 REM fill a triangle with colour
2050 MOVE A,B
2060 MOVE C,D
2070 PLOT 85,E,F
2100 ENDPROC
```

Description

Part of the **DEF PROC...ENDPROC** structure.

Syntax

ENDPROC

Associated keywords

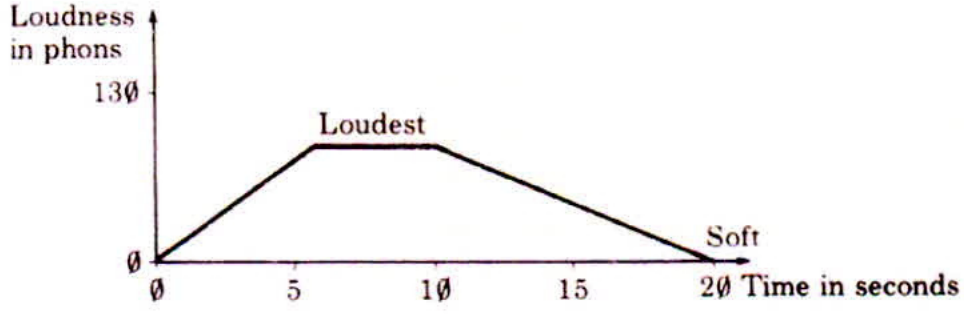
DEF, FN, PROC, LOCAL

ENVELOPE

Purpose

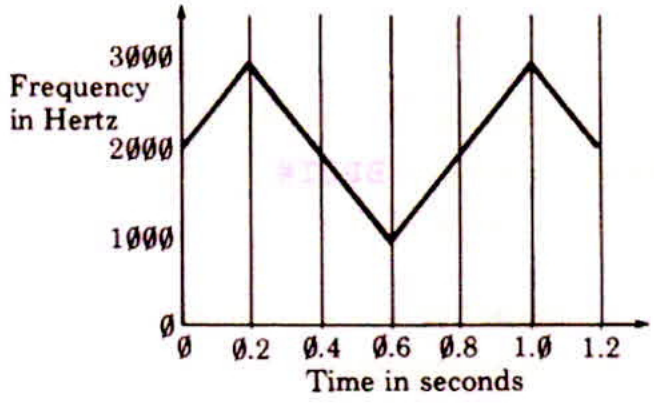
The **ENVELOPE** statement is used with the **SOUND** statement to control the volume and pitch of a sound while it is playing. All natural sounds change in volume (loudness or amplitude); for example, the sounds from a piano start off loudly and then fade away. An aircraft flying overhead starts off softly, gets louder and then fades away.

The variation of amplitude (loudness) for the aircraft, as it flies overhead, looks something like this:



This variation of amplitude with time is described as an 'amplitude envelope'.

Some sounds change in pitch. For example, a wailing police siren:



This variation of pitch with time is called a 'pitch envelope'.

The BBC Microcomputer can use both pitch and amplitude envelopes and these are set up with the **ENVELOPE** statement.

Example

```
10 ENVELOPE 1,1,4,-4,4,10,20,10,127,0,0,-5,126,126
20 SOUND 1,1,100,200
```

Description

The `ENVELOPE` statement is followed by 14 parameters.

```
ENVELOPE N,T,PI1,P12,PI3,PN1,PN2,PN3,AA,AD,AS,AR,ALA,ALD
```

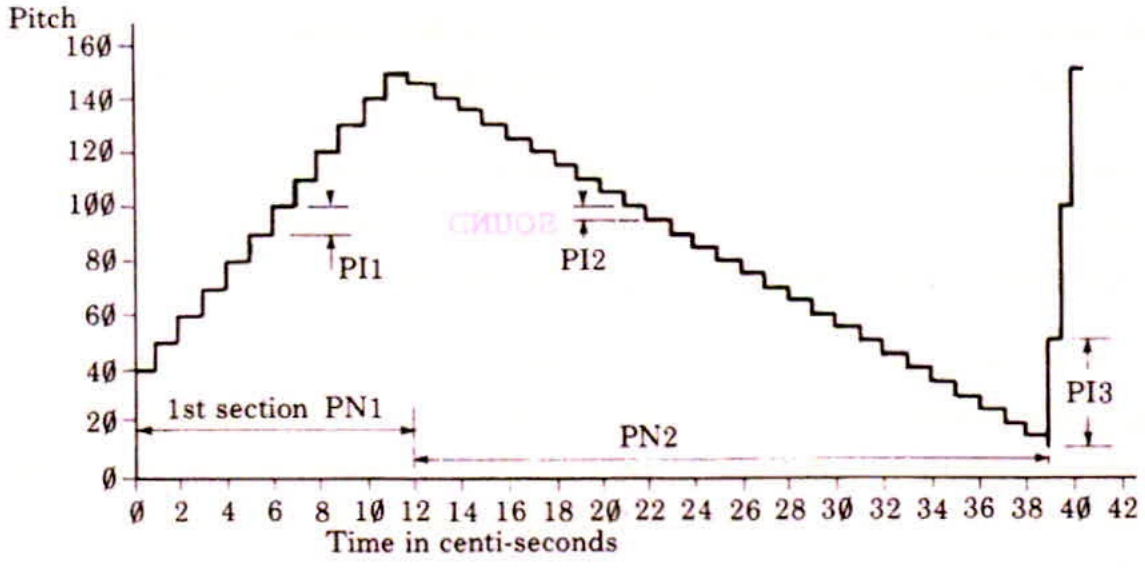
Parameter	Range	Function
N	1 to 4	Envelope number
T bits 0-6	0 to 127	Length of each step in hundredths of a second
T bit 7	0 or 1	0 = auto-repeat pitch envelope 1 = don't auto-repeat pitch envelope
PI1	-128 to 127	Change of pitch per step in section 1
PI2	-128 to 127	Change of pitch per step in section 2
PI3	-128 to 127	Change of pitch per step in section 3
PN1	0 to 255	Number of steps in section 1
PN2	0 to 255	Number of steps in section 2
PN3	0 to 255	Number of steps in section 3
AA	-127 to 127	Change of amplitude per step during attack phase
AD	-127 to 127	Change of amplitude per step during decay phase
AS	-127 to 0	Change of amplitude per step during sustain phase
AR	-127 to 0	Change of amplitude per step during release phase
ALA	0 to 126	Target level at end of attack phase
ALD	0 to 126	Target level at end of decay phase

The **N** parameter specifies the envelope number that is to be defined. It normally has a value in the range 1 to 4. If the **BASIC** statement `BPUT#` is not being used then envelope numbers up to and including 16 may be used.

The **T** parameter determines the length in hundredths of a second of each step of the pitch and amplitude envelopes. The pitch envelope normally auto-repeats but this can be suppressed by setting the top bit of **T** – i.e. using values of **T** greater than 127.

The six parameters **PI1**, **P12**, **P13**, **PN1**, **PN2** and **PN3** determine the pitch envelope. The pitch envelope has three sections and each section is specified with two

parameters: the increment, which may be positive or negative, and the number of times the increment is to be applied during that section, that is the number of steps. A typical pitch envelope might look like:



In the above example T = one hundredth of a second.

PI1 = +10 **PN1 = 12**
PI2 = -5 **PN2 = 27**
PI3 = +50 **PN3 = 3**

The pitch envelope is added to the pitch parameter (P) given in the **SOUND** statement. In the above example it must have been 40 since the pitch starts at 40. If bit 7 of the **T** parameter is zero then at the end of the pitch envelope, at a time given by the equation:

time = (PN1 + PN2 + PN3)*T hundredths of a second

the pitch envelope will be set to zero and will repeat automatically. Note that the pitch can only take on values in the range 0 to 255 and values outside this range are treated as **MOD 256** of the value calculated.

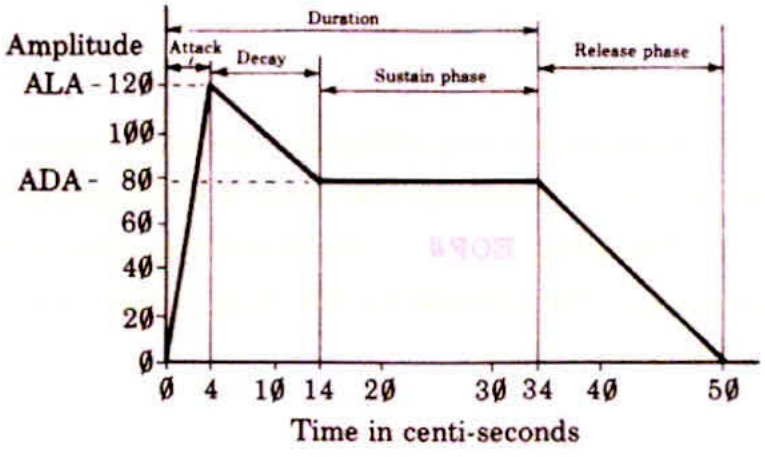
The six parameters **AA**, **AD**, **AS**, **AR**, **ALA** and **ALD** determine the amplitude envelope. Although the current internal sound generator has only 16 amplitude levels the software is upward compatible with a generator having 128 levels.

The shape of the amplitude envelope is defined in terms of rates (increments) between levels, and is an extended form of the standard ADSR system of envelope control. The envelope starts at zero and then climbs at a rate set by **AA** (the attack rate) until it reaches the level set by **ALA**. It then climbs or falls at the rate set by **AD** (the decay rate) until it reaches the level set by **ALD**. However, if **AD** is zero the amplitude will stay at the level set by **ALA** for the duration (D) of the sound.

The envelope then enters the sustain phase which lasts for the remaining duration (D) of the sound. The duration is set by the **SOUND** statement. During the sustain phase the amplitude will remain the same or fall at a rate set by AS.

At the end of the sustain phase the note will be terminated if there is another note waiting to be played on the selected channel. If no note is waiting then the amplitude will decay at a rate set by AR until the amplitude reaches zero. If **AR** is zero then the note will continue indefinitely, with the pitch envelope auto repeating if bit 7 of parameter **T** is zero.

A typical amplitude envelope might look like:



In the above example T = one hundredth of a second.

ALA = 120

ADA = 80

AA = 30 (120 in four hundredths of a second)

AD = -4 (-40 in 10 hundredths of a second)

AS = 0

AR = -5 (-80 in 16 hundredths of a second)

Note that the amplitude cannot be moved outside the range 0 to 126.

Syntax

ENVELOPE <numeric>,<numeric>,<numeric>,<numeric>,<numeric>,
<numeric>,<numeric>,<numeric>,<numeric>,<numeric>,<numeric>,
<numeric>,<numeric>,<numeric>

Associated keywords

ADVAL, SOUND

EOF#

Purpose

This function is used to determine whether the end of the file has been reached or not. The function returns the value 0 or -1. It returns the value -1 if the end of the file has been reached. The number following `EOF#` is the channel number of the file. See the chapter on file handling in the manual for the filing system you are using.

Examples

```
100 X=EOF# (channel)
200 REPEAT UNTIL EOF#(y)
```

Description

The function used to determine whether the end of the file has been reached or not.

Syntax

<num-var>=**EOF#**(<num-var>)

Associated keywords

OPENIN, OPENUP, OPENOUT, EXT#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, CLOSE#

EOR Exclusive-OR

Purpose

This is a special logical operator often used to complement certain bits in a byte selectively. Refer to the keyword **AND** for an introduction to the concepts involved.

The process of Exclusive-OR tests whether the corresponding bits in two numbers are the same or different. If the corresponding bits in the two numbers are different then the resultant bit will be a one, if they are the same it will be a zero.

Another way of looking at this process is that it complements (changes 0 to 1 and 1 to 0) those bits in one number which are at logic 1 in the other number. Thus if:

X is	0000 1100 0011 0000 1110 1011
Y is	1011 1111 0000 1010 0010 1000

Then

X **EOR** Y is	1011 0011 0011 1010 1100 0011

Examples

```
100 d%= A% EOR &FFFF00
200 R= X EOR Y
```

Description

An operator performing the operation of logical or bitwise Exclusive-OR between the two operands.

Syntax

<num-var>=<numeric>**EOR**<numeric>

or

<testable condition>=<testable condition>**EOR**<testable condition>

Associated keywords

NOT, AND, OR

ERL error line number

Purpose

This enables the program to find out the line number where the last error occurred.

Examples

```
8500 X=ERL
8100 IF ERL=100 THEN PRINT "I didn't understand"
300 IF ERL=10000 THEN CLOSE#0
```

Description

A function returning the line number of the line where the last error occurred.

Syntax

<num-var>=ERL

Associated keywords

ON ERROR..., ON ERROR OFF, REPORT, ERR

ERR error

Purpose

If the computer finds an error that it cannot resolve, it may give up and report the error on the screen. In addition it remembers an 'error number'. For example, if you try to calculate with numbers which are too large for the computer it will report `Too big` and remember error number 20.

Pressing the **<ESCAPE>** key behaves as an error (error number 17) and you can detect this and act on it if you wish.

It is possible to make the computer deal with most of these errors itself by writing special sections of the program to deal with the inevitable! These sections of the program need to know what the error was and where it occurred.

The function `ERR` enables your program to find the error number of the last error which occurred. This is generally used to enable the program to respond helpfully to an error caused by the user. See Chapter 6 for a full list of error messages and numbers found in **BASIC**.

Examples

```
1000 wrong=ERR
100 IF ERR=17 THEN PRINT "YOU CAN'T ESCAPE!"
1230 IF ERR=18 THEN PRINT "You can't divide by zero!"
```

Description

Returns the error number of the last error which occurred.

Syntax

<num-var>=`ERR`

Associated keywords

ON ERROR..., ON ERROR OFF, ERL, REPORT

EVAL evaluate

Purpose

This function is mainly used to enable the user to type an expression, such as a mathematical equation, into the computer while a program is running.

For example, suppose that a program has to plot a graph; you need a way of getting your equation into the computer while a program is running. In most versions of **BASIC** this is very difficult to do. With the **BBC BASIC** the equation is put into a string and then `EVAL` is used to tell the computer to work out the string.

The function is not common in other versions of **BASIC** so a few more specific examples are given of instructions which can be evaluated by the statement `EVAL(A$)`.

```
A$="M*X+C"
A$="SIN(x/120)+COS(x/30)"
```

Note that `EVAL` can be used to evaluate functions (`SIN, COS, SQR` etc.) but cannot be used to execute a statement like `MODE 4`.

Examples

```
100 X=EVAL(A$)
234 value=EVAL (z$)
```

Description

A statement which applies the interpreter's expression evaluation program to the characters held in the argument string. An easy way to pass a function into a program from a user input.

Syntax

<num-var>=**EVAL**(<string>)

or

<str-var>=**EVAL**(<string>)

Associated keywords

VAL, STR$

Demonstration program

```
10 INPUT A$
20 FOR X=1 TO 5
30 Y=EVAL(A$)
40 PRINT Y
50 NEXT X
>RUN
?5*X
5
10
15
20
25
```

This second program makes the computer act as a calculator.

```
 5 REPEAT
10 INPUT B$
20 PRINT EVAL B$
30 UNTIL FALSE
>RUN
?3+4
7
?SIN (RAD (45))
0.707106781
?
```

EXP exponent

Purpose

This mathematical function calculates e (2.7183...) raised to any specified power.

Examples

```
120 Y=EXP(X)
3000 pressure=EXP(height)
```

Description

A function returning e to the power of its argument.

Syntax

<num-var>=**EXP**(<numeric>)

Associated keywords

LN, LOG

EXT# extent

Purpose

This function finds out how large a particular file is. It only works with mass media storage systems, such as floppy and hard disc, and Econet filing systems – not with cassette files. The number returned is the number of bytes allocated to the file. The file to be investigated must have been opened during the **OPENIN, OPENOUT** or **OPENUP** statements. See the chapter on file handling in the manual for the filing system you are using.

Examples

```
100 X=EXT#(employee)
PRINT EXT# (N)
```

Description

A function which returns the length in bytes of the file opened on the channel given in its argument.

Syntax

<num-var>=**EXT#**(<num-var>)

Associated keywords

CLOSE#, PTR#, INPUT#, PRINT#, BGET#, BPUT#, OPENIN, OPENUP, OPENOUT, EOF#

FALSE

Purpose

Sometimes the computer has to decide whether or not something is true. For example:

```
10 X=12
20 IF X=20 THEN PRINT "X EQUALS 20"
```

Clearly, in this example, the statement `x=20` is false. As program will never print `X EQUALS 20.`

```
100 X=12
110 REPEAT
120 PRINT "HELLO"
130 UNTIL X=20
```

would repeatedly print `HELLO` because X will never be anything other than 12. `X=20` is `FALSE`. The same effect can be achieved by writing:

```
110 REPEAT
120 PRINT "HELLO"
130 UNTIL FALSE
```

which means repeat for ever.

In fact, the computer has a numerical value of **FALSE**, which is zero. Thus:

```
PRINT FALSE
```

will print `0`.

Similarly:

```
PRINT 5=4
```

will also print `0`, since `5=4` is false.

It is often useful to say in a program for example:

```
CLOCKSET = FALSE
```

and then later you can say:

```
IF CLOCKSET THEN PRINT "THE CLOCK IS CORRECT"
```

Examples

```
100 oldenough = FALSE
245 UNTIL FALSE
```

Description

A function returning the value zero.

Syntax

<num-var>=**FALSE**

Associated keywords

TRUE

FN function

Purpose

FN preceding a variable name indicates that it is being used as the name of a function. Both string and numeric functions may be defined. See the keyword **DEF** for a more detailed description of functions and procedures.

Since a function always returns a value. It often appears on the right of an equal sign or in a print statement. Procedures, on the other hand, do not return results.

Example

```
1000 DEF FNmean2(x,y)=(x+y)/2
```

Description

A reserved word used at the start of all user defined functions.

Syntax

DEF FN<variable-name>[(<num-var>|<str-var>{,<num-var>|<strvar>})]

Associated keywords

ENDPROC, DEF, LOCAL

FOR

Purpose

The word **FOR** is one of the words used in a **FOR...NEXT** loop. This makes the computer execute a series of statements a specified number of times; for example:

```
120 FOR X=1 TO 5
130 PRINT X
140 NEXT X
```

would print out the numbers **1, 2, 3, 4, 5.**

The variable **X** in the above example initially takes on the value 1 and the program then goes through until it reaches the word **NEXT**. The program then returns to the line or statement **FOR X=1 TO 5** and **X** is increased in value by 1. The program continues the loop, increasing the value of **X** in steps of 1, until **X** reaches 5. After that, the program no longer loops; instead it moves onto the next statement after the **FOR...NEXT** loop.

As an option the 'step size' can be changed. In the example above **X** increased by 1 each time around the loop. In the next example **XYZ** increases by 0.3 each time around the loop.

```
230 FOR XYZ=5 TO 6 STEP 0.3
240 PRINT XYZ
250 NEXT XYZ
```

The above program would print out the numbers

```
5
5.3
5.6
5.9
```

The value of XYZ on exit from the above program would be 6.2. The step size may be negative if you wish to make the value of the control variable decrease each time around the loop.

```
870 FOR r2d2%=99 TO 60 STEP -12
880 PRINT r2d2%;"Hi there"
890 NEXT r2d2%
```

would print

```
99Hi there
87Hi there
75Hi there
63Hi there
```

The FOR...NEXT loop always executes once so:

```
FOR D=5 TO 3: PRINT D: NEXT D
```

would print 5 and then stop.

Examples

```
300 FOR X=1 TO 16 STEP 0.3: PRINT X: NEXT X
1040 FOR A%=0 TO MAXIMUM%
560 FOR TEMPERATURE=0 TO 9
```

Description

A statement initialising a FOR...NEXT loop. This structure always executes at least once. Any assignable numeric item may be used as the control variable but integer control variables are about three times faster than real variables. Note that when the step is non-integer, rounding errors may creep in and the value of the control variable may well diverge significantly from the true arithmetic value.

Syntax

FOR <num-var>=<numeric>TO<numeric>[STEP<numeric>]

Associated keywords

TO, STEP, NEXT

GCOL

Purpose

This statement sets the colour to be used by all subsequent graphics operations. In other words it selects the graphics foreground colour and the graphics background colour. It also specifies how the colour is to be placed on the screen. The colour can be plotted directly, ANDed, ORed or Exclusive-ORed with the colour already there, or the colour there can be inverted (reversed).

The first number specifies the mode of action as follows:

0. Plot the colour specified,
1. OR the specified colour with that already there,
2. AND the specified colour with that already there,
3. Exclusive-OR the specified colour with that already there,
4. Invert the colour already there.

The second number defines the logical colour to be used in future. If the number is greater than 127 then it defines the graphics background colour. If the number is less than 128 then it defines the graphics foreground colour. See the keyword `COLOUR` for more information.

Examples

```
100 GCOL 0,2
GCOL 3,129
```

Description

This statement is used to select the logical colours used by graphics statements.

Syntax

`GCOL`<numeric>`,`<numeric>

Associated keywords

CLS, CLG, MODE, COLOUR, PLOT

GET

Purpose

This function waits for a key to be pressed on the keyboard and then returns with the ASCII number of the key pressed.

The **GET** function is used whenever the computer needs to wait for a reply from the user before continuing.

Note that when using **GET** the character typed on the keyboard will not appear on the screen. If you wish it to appear you must then ask the computer to print it.

Examples

```
1040 keyhit=GET
350 X=GET
```

Description

A function which waits for the next character from the input stream. The function then returns the ASCII value of the character.

Syntax

<num-var>=**GET**

Associated keywords

GET$, INKEY, INKEY$

GET$

Purpose

The GET$ function waits for a key to be pressed on the keyboard and then returns with a string containing the character pressed. See the previous keyword, GET, for a similar function and for further explanation.

For example, at the end of a game program you may wish the computer to ask the player whether or not he or she wants to go again. The demonstration program below shows how this can be done.

Note that when using GET$ the character typed on the keyboard will not appear on the screen. If you wish it to appear you must ask the computer to print it.

Example

```
1050 A$=GET$
```

Syntax

<string-var>=GET$

Associated keywords

GET, INKEY, INKEY$

Demonstration program

```
2110 PRINT "Do you want to play another game";
2120 REM if user presses Y then
2130 REM go to line 100
2140 IF GET$="Y" THEN GOTO 100
2160 REM if it gets this far then the
2170 REM reply was not "Y" so give up!
2180 STOP
```

GOSUB go to a subroutine

Purpose

Often a group of lines in a program needs to be used in a number of different places within the main program. Instead of repeating the same piece of program several times you can make this small sub-section into a separate subroutine. This subroutine can then be 'called' from a number of different places in the main program by means of the statement **GOSUB**. The end of a subroutine is indicated by the word **RETURN**. This causes the program to return to the statement after the **GOSUB** statement.

Beware of a subroutine calling itself too many times: a depth of 26 subroutines is the maximum that is allowed.

As with **GOTO**, it is possible to **GOSUB** to a calculated line number. The same cautions that apply to **GOTO** apply to **GOSUB** in this case.

Example

```
1020 GOSUB 4000
```

Description

A statement used to call a section of program as a subroutine. One subroutine may call another subroutine (or itself) up to a maximum nested depth of 26.

Syntax

GOSUB<numeric>

ON<num-var>**GOSUB**<numeric>{,<numeric>}[**ELSE**<statement>]

Associated keywords

RETURN, ON

Demonstration program

First, here is a program to print out random phrases without using a subroutine:

```
100 REM A$ contains 7 words and each word
105 REM contains 5 characters - letters or spaces
110 A$="hand mouthear leg arm chestelbow"
120 FOR count=1 TO 10
125 REM pick a random number
130 R=RND(7)
140 REM and use it to pick a random word
150 B$=MID$(A$,5*R-4,5)
160 REM print a message
170 PRINT "My ";B$;" hurts"
180 REM get another random word
190 R=RND(7)
200 B$=MID$(A$,5*R-4,5)
210 REM and print out a second message
220 PRINT "Is your ";B$;" all right?"
230 NEXT count
```

Now look at the same program using a subroutine and with the **REM**s (remarks) removed.

```
110 A$="hand mouthear leg arm chestelbow"
120 FOR count=1 TO 10
150 GOSUB 810
170 PRINT "MY ";B$;" hurts"
190 GOSUB 810
220 PRINT "Is your ";B$;" all right?"
230 NEXT count
240 END
810 B$=MID$(A$,5*RND(7)-4,5)
820 RETURN
```

The 7 in line 810 is there to select one of the seven words in A$. In line 810 both the 5s are there because each word contains five letters or spaces. It is essential that all the words contain the same number of characters.

Finally, here is the same program written with a string function, with **REM**s left out and generally cleaned up.

```
110 A$="hand mouthear leg arm chestelbow"
120 FOR count=1 TO 10
170 PRINT "My ";FNword;" hurts"
220 PRINT "Is your ";FNword;" all right?"
230 NEXT count
240 END
800 DEF FNword= MID$(A$,5*RND(7)-4,5)
```

GOTO go to a line number

Purpose

This statement makes the computer jump to a specified line number instead of continuing to the next one in the program. It changes the order in which the computer executes a program.

Although `GOTO` is simple to use, do so with caution! It is all too easy to make a program difficult to follow by using too many `GOTO`s. Following a program full of `GOTO`s is like trying to disentangle a plateful of spaghetti and arrange it in a straight line!

Adherents of 'structured programming' encourage program writers to use structures like `REPEAT...UNTIL` and `FOR...NEXT` and to avoid most (but not all) `GOTO` statements.

It is possible in this version of **BASIC** to `GOTO` a variable. In the following example the destination variable is called 'somewhere':

```
10 somewhere=1005
20 GOTO somewhere
```

but this feature must be used with great care since the program cannot be renumbered using the `RENUMBER` command.

Note that if the destination line number is to be calculated using a mathematical expression then that expression must be in parentheses.

`GOTO` can be used as a command to start a program without destroying the values assigned to the variables.

Examples

```
GOTO 330
100 IF X>5 THEN GOTO 2000
100 GOTO (starts*55+14)
```

Description

A statement used to transfer control to a specified or calculated line number unconditionally.

Syntax

GOTO<numeric>

ON<num-var>**GOTO**<numeric>{,<numeric>}[**ELSE**<statement>]

HIMEM highest memory location

Purpose

BASIC uses the computer's random access memory (RAM) to store the user's program, all the variables that the program uses, and memory for high-resolution graphics displays.

In the absence of other instructions the computer divides the available memory up logically. However there are occasions, particularly when changing display modes and when writing machine code programs, when you may wish to tell **BASIC** how to divide up the available memory.

One way of changing the allocation is by altering the value of the variable `HIMEM`. This variable contains the address of the highest memory location that **BASIC** uses for your program and variables. It is automatically set to just below the memory used for the screen when the `MODE` is selected. Addresses above `HIMEM` are not used by **BASIC**.

If it is manually altered then locations above `HIMEM` may be used by the programmer for other things, for example for machine code subroutines.

If you wish to change the value of `HIMEM` you should normally do so very early in your program – preferably right at the beginning. The beginning of the program is also the place to select the display mode that you will be using.

Other important boundaries are `PAGE`, `TOP` and `LOMEM`.

Note that in the 'shadow screen' mode used on the Model B+, `HIMEM` always returns a value of &8000.

Examples

```
100 HIMEM=HIMEM-40
100 PRINT HIMEM
100 HIMEM = &2800
```

Description

HIMEM contains the address of the first byte that **BASIC** does not use. This pseudo-variable must not be altered while executing a function or a procedure. Alter it with great care! When using a second processor, or if the computer is being used in shadow mode, **HIMEM** will not be altered when changing **MODE**.

Syntax

HIMEM=<numeric>

or

<num-var>=**HIMEM**

Associated keywords

LOMEM, PAGE, TOP

IF

Purpose

This sets up a test condition which can be used to control the subsequent action of the computer.

Examples

```
100 IF month=12 THEN PRINT "December"
100 IF A=1 THEN PRINT "One" ELSE PRINT "Not one"
100 IF answer$="BANANA" THEN PROCfruit
100 IF height<1.94 OR age<18 THEN GOTO 1030
100 IF length <>5 THEN 2140
100 IF RATE=5 THEN Y=6:Z=8 ELSE PRINT "Wrong rate"
100 IF month=11 THEN IF day=5 THEN PRINT "Guy Fawkes"
100 IF month=1 AND day=1 THEN PRINT "New Year"
100 IF X THEN Y=0
```

Description

A statement forming part of the `IF...THEN...ELSE` structure. The word `THEN` is optional as is the `ELSE` section.

Syntax

`IF`<testable condition>[`THEN`]<statement | numeric>[`ELSE`<statement | numeric>]

Associated keywords

THEN, ELSE

INKEY input the number of the key pressed

Purpose

This function waits for a specified time while constantly testing to see if a key has been pressed on the keyboard.

If a key is pressed before the time runs out then the ASCII value of the key is given. If no key is pressed in the given time then -1 is returned and the program continues. See the keyword **ASC** for an explanation of ASCII values.

Note that a key can be pressed at any time before **INKEY** is used. All keys pressed are stored in a buffer in the computer and a character is removed from the buffer by, for example, the **INPUT** statement. You can clear the buffer of all characters by giving the command:

```
*FX 15,1
```

The number in parenthesis, after the word **INKEY**, gives the amount of time that the computer must wait before giving up. The time is given in hundredths of a second, and may have any value between 0 and 32767.

In addition, the function **INKEY** can be used to see if a key is actually pressed at the instant the function is called. Normally pressing a key once enters the code for that key into the keyboard buffer. If the key is kept down then it will normally auto-repeat and further characters will be entered into the buffer. However, when the buffer is read with **INPUT** or **GET** or **INKEY**, you will have no idea how long the character has been waiting in the buffer. An alternative statement is provided which actually tests the keyboard rather than the buffer.

INKEY with a negative number in the parenthesis, e.g. **INKEY(-27)** will enable you to test to see whether a particular key is pressed at that instant. The number in parenthesis determines which key you wish to test. The following table shows the negative number to be used to test any particular key. Thus the letter L would be tested with **PRINT INKEY (-87).**

Examples

```
100 keynumber=INKEY(5)
220 result=INKEY(Y)
X=INKEY(100)
```

Description

A function which waits up to a specified time for a key to be pressed. The function returns -1 if no key is pressed in the specified time, or the ASCII value of the key pressed. The argument is the maximum time in hundredths of a second.

Syntax

<num-var>=**INKEY**(<numeric>)

Associated keywords

GET, GET$, INKEY$

Key	Number	Key	Number
f0	-33	1	-49
f1	-114	2	-50
f2	-115	3	-18
f3	-116	4	-19
f4	-21	5	-20
f5	-117	6	-53
f6	-118	7	-37
f7	-23	8	-22
f8	-119	9	-39
f9	-120	0	-40
A	-66	-	-24
B	-101	^	-25
C	-83	\	-121
D	-51	@	-72
E	-35	[	-57
F	-68	–	-41
G	-84	;	-88
H	-85	:	-73
I	-38	]	-89
J	-70	‘	-103
K	-71	.	-104
L	-87	/	-105
S	-82	Space bar	-99
M	-102	**ESCAPE**	-113
N	-86	**TAB**	-97
O	-55	**CAPS LOCK**	-65
P	-56	**CTRL**	-2
Q	-17	**SHIFT LOCK**	-81
R	-52	**SHIFT**	-1
T	-36	**DELETE**	-90
U	-54	**COPY**	-106
V	-100	**RETURN**	-74
W	-34	↑	-58
X	-67	↓	-42
Y	-69	←	-26
Z	-98	→	-122

INKEY$ input the character pressed

Purpose

This function waits for a specified time while constantly testing to see if a key has been pressed on the keyboard. If a key is pressed before the time runs out then the letter or number pressed is placed in the string variable. If no key is pressed in the given time then an empty string is returned and the program continues.

Note that a key can be pressed at any time before **INKEY$** is used. All keys pressed are stored in a buffer in the computer and a character is removed from the buffer by, for example, the **INPUT** statement. You can clear the buffer of all characters by giving the command:

```
*FX 15,1
```

The number in parenthesis, after the word **INKEY$**, gives the amount of time that the computer must wait before giving up. The time is given in hundredths of a second.

Examples

```
120 letter$=INKEY$(0)
384 result$=INKEY$(100)
920 X$=INKEY$(Y)
```

Description

A function which waits for a key to be pressed within a specified period of time. The function returns a null string if no key is pressed in the specified time. If a key is pressed the string returned consists of the single character pressed. The argument is the maximum time in hundredths of a second.

Syntax

<string-var>=**INKEY$**(<numeric>)

Associated keywords

GET, GET$, INKEY

INPUT to put information into the computer

Purpose

When a computer program is running there is often a need to get numbers or words from the outside world into the computer so that it can do calculations on these numbers or words. The statement `INPUT` is used for this purpose. There are a number of options:

```
100 INPUT X
```

will print a question mark on the screen and wait for the user to type in a number. This is not very 'friendly' – often it would be helpful to print a message on the screen before waiting for the user to type his/her reply. This can be done in two ways:

```
340 PRINT "How old are you";
350 INPUT AGE
```

or more simply:

```
340 INPUT "How old are you",AGE
```

If you do not wish the computer automatically to print a question mark then omit the comma between the message to be printed out and the variable to be filled in:

```
340 INPUT "How old are you" AGE
```

Often you may want to input several values one after the other. This can be done by placing the variables after each other, but separated by commas, thus:

```
560 INPUT "Pick three numbers",X,Y,Z
```

When replying the user separates the values entered either with commas or by pressing the **<RETURN>** key after entering each value. The numbers that are typed in are placed in the appropriate variables – `X`, `Y` and `Z` in the example above.

The above examples all required numbers to be supplied by the user. You can **INPUT** words as well:

```
250 INPUT "What is your name",NAME$
```

You can **INPUT** more than one string at a time if you wish by using:

```
200 INPUT "Town",A$,"Country",B$
```

INPUT LINE A$** will accept everything that is typed in including leading spaces and commas, and will place everything into **A$.

INPUT may be used with **TAB** in the same way that **PRINT** can be. For example:

```
300 INPUT TAB(3,12) "number please" X
```

SPC can be used also, to insert spaces.

A semi-colon can replace a comma. Either a semi-colon or a comma will cause the computer to print a question mark when waiting for the variable to be filled.

Description

A statement to input values from the current input stream. The question mark prompt may be suppressed by omitting the comma following the prompt string. **INPUT** strips leading spaces off strings.

Syntax

INPUT[<string-const>][**;** | **,**]<num-var> | <string-var>{**,** <numvar> | <string-var>}

Associated keywords

INPUT#, LINE, TAB, SPC

INPUT# put information into the computer from a filing system

Purpose

It is possible to record data (numbers and words) on a storage medium where they can be stored for later use. The statement **INPUT#** is used to read the data back into the computer. See the chapter on file handling in the manual for the filing system you are using.

Examples

```
1200 INPUT# channel, date, name$, address$
3400 INPUT#X,U,V,W$
```

Description

A statement which reads data in internal format from a file and places the data in the stated variables.

Syntax

INPUT#<num-var>,<num-var>|<string-var>{,<num-var>|<stringvar>}

Associated keywords

OPENIN, OPENUP, OPENOUT, EXT#, PTR#, PRINT#, BGET#, BPUT#, CLOSE#

INSTR in string

Purpose

To search one string for any occurrence of another string, for example to see if one word contains another specific word. The search normally starts from the beginning of one string but as an option the search can start from a specified point along the string.

The number returned is the string position of the second string in the first string. The leftmost character position is position number 1. If no match is found then zero is returned. A search for a null string:

```
X=INSTR("Sunday","")
```

will always return 1.

Examples

```
240 X=INSTR(A$,B$,Z)
```

puts the position of `B$` in `A$` into `X`, starting the search as position `Z`.

```
PRINT INSTR("HELLO","L")
```

would print `3`.

Description

A function which returns the position of a sub-string within a string. The starting position for the search may be specified. There must be no space between `INSTR` and the first parenthesis.

Syntax

<num-var>=**INSTR**(<string>,<string>[,<numeric>])

Associated keywords

LEFT$, MID$, RIGHT$, LEN$

INT integer part

Purpose

This converts a number with a decimal part to a whole number. This function always returns a whole number smaller than the number supplied. Thus `INT(23.789)` gives 23 whereas `INT(-13.3)` returns -14.

Examples

```
200 X = INT(Y)
1050 wholenumber=INT (decimalnumber)
330 pence=INT(cost * markup/quantity)
```

Description

`INT` is a function converting a real number to the lower integer.

Syntax

<num-var>=**INT**<numeric>

LEFT$ left string

Purpose

To copy part of a string starting at the left of the source string. For example if:

```
A$="CATASTROPHE"
```

then:

```
PRINT LEFT$(A$,3)
```

would give `CAT`, namely the left three characters of `A$`.

Examples

```
100 INDEX$=LEFT$(WHOLEname$,4)
3000 U$=LEFT$(H4$,value)
```

Description

A string function which returns the left n characters from a string. If the source string is too short then the function returns with as many characters as there are in the source string. There must be no space between `LEFT$` and the first parenthesis.

Syntax

<string-var>=`LEFT$`(<string>,<numeric>)

Associated keywords

RIGHT$, MID$, LEN, INSTR

Demonstration program

This prints out letters in a pattern.

```
10 PRINT "What is your full name";
20 INPUT name$
30 FOR X = 1 TO LEN(name$)
40 PRINT LEFT$(name$,X)
50 NEXT X
>RUN
253
What is your full name?JOHN A COLL
J
JO
JOH
JOHN
JOHN
JOHN A
JOHN A
JOHN A C
JOHN A CO
JOHN A COL
JOHN A COLL
```

LEN length (of a string)

Purpose

This function counts the number of characters in a string. For example:

```
K=LEN("FRIDAY ")
```

would give `K=7` since there are six letters in `"FRIDAY"` and it is followed by a space.

This function is often used with a `FOR...NEXT` loop to do something once for each letter in a string. For example, we might wish to encode a word by replacing each letter with its successor in the alphabet so that, for example, `"FRIDAY"` would become `"GSJEBZ"`. See the demonstration program below.

Examples

```
100 X=LEN(A$)
2350 length=LEN(main$)
```

Description

This function returns the length of the string given as the argument.

Syntax

<num-var>=**LEN**(<string>)

Associated keywords

LEFT$, MID$, RIGHT$, INSTR

Demonstration program

```
300 PRINT "Type in your word";
310 INPUT A$
320 Length=LEN(A$)
325 C$=""
330 FOR V=1 TO length
340 B$=MID$(A$,V,1)
350 C$=C$+CHR$(ASC(B$)+1)
360 NEXT V
370 PRINT "The coded version is ";C$
```

In the above program each letter is copied one at a time into **B$**. Then its ASCII value is calculated, 1 is added to the ASCII value and the new ASCII value is converted back into a character which is then added onto **C$**. See the keyword **ASC** for more information about the ASCII code.

LET

Purpose

In **BASIC** we often write things like:

```
X=6
```

meaning put `6` into the box labelled `X` in the computer. The fact that we are changing the contents of the variable `X` can be made clearer by writing:

```
LET X=6
```

The statement `X=X+6` is impossible in mathematical terms. How can something be the same as itself plus 6? In **BASIC** though it is quite legal to say `LET X=X+6` since the instruction simply means 'store in the variable `X` whatever is already there plus `6`', or 'increase the value of `X` by `6`'.

The word `LET` is optional, but its use makes the program more readable.

Examples

```
100 LET length=15
980 LET DAY$ ="Tuesday"
210 IF A=6 THEN LET length=12
```

Description

`LET` is an optional assignment statement.

NOTE: `LET` may not be used during the assignment of the pseudo-variables LOMEM, HIMEM, PAGE, PTR#, TIME.

Syntax

[`LET`]<var>=<expression>

LIST

Purpose

This command makes the computer list whatever program it has in its memory. It is often used before typing `RUN` to ensure that there aren't any typing errors in the program just entered.

You can list a single line:

```
LIST 280
```

or a range of lines:

```
LIST 100,450
```

or the whole program:

```
LIST
```

`LIST ,400` will list all lines up to and including line 400.

`LIST 400,` will list all lines beyond line 400.

If you have a very long program you may see the whole listing whiz past before you have time to read it. To stop it, and to make the computer stop at the bottom of each page you can type `<CTRL>` `<N>` (while holding down the key marked **CTRL**, press the letter **N**). Then type `LIST`. This is called 'page mode' and the computer stops at the bottom of each page. The next page will be printed when the `<SHIFT>` key is pressed.

To return to 'scroll mode' type `<CTRL>` `<O>` (hold **CTRL** down while briefly pressing **O**). Pressing `<CTRL>` and `<SHIFT>` together immediately pauses a listing, and pressing `<ESCAPE>` will stop a listing so that corrections can be made to the program.

If you want a listing on the printer then you can turn the printer on by typing `<CTRL>` `<B>` before typing `LIST`.

To turn the printer off afterwards type <CTRL> <C>.

LIST is a command and cannot be used as part of a program or as part of a multiple statement line.

The layout of programs as listed can be controlled by the command **LISTO** (see next entry). As an option, the computer can be instructed to insert spaces for the duration of all **FOR...NEXT** and **REPEAT...UNTIL** loops.

Examples

```
LIST
LIST 400
LIST 400,500
LIST ,900
LIST 900,
```

Description

A command which lists the current program.

Syntax

LIST[,][<num-const>][,][<num-const>]

Associated keywords

NEW, OLD, LISTO

LISTO list option

Purpose

When a program is listed on the screen or the printer, it is often convenient to show all loops within the program indented. `LISTO` can be used to control the way that the `LIST` command displays a program on the screen. It can cause the computer to insert spaces in three situations:

- After the line number.
- During `FOR...NEXT` loops.
- During `REPEAT...UNTIL` loops.

The number following `LISTO` should be in the range 0 to 7:

0 - Implies no inserted spaces.
1 - Implies a space after the line number.
2 - Implies spaces during `FOR...NEXT` loops.
4 - Implies spaces during `REPEAT...UNTIL` loops.

The numbers which select each option (1, 2 or 4) can be added together to select multiple options. If spaces were required during `FOR...NEXT` and `REPEAT...UNTIL` loops then `LISTO6` would be selected. `LISTO7` puts a space after the line number and double spaces for `FOR...NEXT` and `REPEAT...UNTIL` loops.

When editing programs using the cursor editing keys it is strongly advised that you use the `LISTO0` option or else you will **COPY** in a lot of extra space.

Description

`LISTO` affects the print format produced by subsequent `LIST` commands.

Syntax

`LISTO`<num-const>

Associated keywords

LIST

LN natural logarithm

Purpose

A mathematical function to calculate logarithms to the base **e** – usually called 'natural logarithms'.

Examples

```
100 X=LN(temp)
3000 H5=LN(REDOXpotential)
```

Description

A function returning the natural logarithm of its argument. Inverse logarithms (anti-logarithms) can be calculated by using

```
antilog = EXP(log)
```

Syntax

<num-var>=**LN**<numeric>

Associated keywords

LOG, EXP

LOAD

Purpose

To load a program into the computer from the current active filing system, be that cassette tape, disc, ADFS, Econet, or any other.

```
LOAD "GAME1"
```

Once the program has been loaded, type `RUN` to start it.

When you use the word `LOAD`, the computer forgets any previous program it had in memory and also the values of all variables.

If you are loading from disc then the file name (enclosed in quotes) must be a string of not more than seven characters in length (or ten characters for the Advanced Disc Filing System). If a disc directory is specified then you do this by putting the directory character before the file name, like this:

```
LOAD "B.GAME1"
```

If you wish to load from a drive other than the one currently selected then the drive number also is included in the quotes preceded by a colon. For example:

```
LOAD ":0.D.GAME1"
```

will load a file called `GAME1` in directory `D` from drive `0`.

If you are loading from cassette, then the computer will show the name of each section of the program as it finds it on the cassette. The file name (enclosed in quotes) may be up to ten characters in length. `LOAD""` (with no file name) will load the next program found on cassette, whatever its name. This does not work on disc or Econet or other filing systems.

`LOAD` does not run a program. It just loads a file into memory. It clears all variables except A% to Z% and @%. The command `LOAD` cannot be used in a program.

The statement **CHAIN** can be used in a program (or as a command) to load another program and to start that program running automatically.

Examples

```
LOAD "STARWARS"
LOAD "MYPROG"
```

Description

The command **LOAD** deletes the current program, clears all variables except the resident integer variables and then loads a new program from the current filing system. The program to be loaded must be in internal format.

Since **LOAD** is a command it cannot form part of a multiple statement line.

Syntax

LOAD<string>

Associated keywords

SAVE, CHAIN

LOCAL

Purpose

This informs the computer that the named variables are 'local' to the procedure or function in which they occur; their use in this procedure or function in no way affects their value outside it. See the keyword **DEF** for more information.

Example

```
560 LOCAL X,Y,A$,B$
```

Description

A statement which can only be used inside a procedure or function definition. **LOCAL** saves the values of the external variables named and restores these original values when the function or procedure is completed.

Syntax

LOCAL<string-var>|<num-var>{,<string-var>|<num-var>}

Demonstration procedure

```
780 DEF PROCdrawTRIANGLE(size)
790 LOCAL X1,X2,Y1,Y2
800 X1=320-size
810 X2=320+size
820 Y1=256-size
830 Y2=256+size
840 MOVE X1,Y1
850 DRAW X2,Y1
860 DRAW 320,Y2
870 DRAW X1,Y1
880 ENDPROC
```

Associated keywords

DEF, ENDPROC, FN, PROC

LOG logarithm

Purpose

A mathematical function to calculate the common logarithm of a number to base 10.

Examples

```
100 Y=LOG(y)
440 pressure=LOG(speed)
```

Description

A function giving the common logarithm to base 10 of its argument. Inverse logarithms (anti-logarithms) can be calculated by using:

```
antilog = 10^log
```

Syntax

<num-var>=LOG<numeric>

Associated keywords

LN, EXP

LOMEM

Purpose

Different sections of the computer's memory are used for different purposes. Normally **BASIC** makes an intelligent decision about where to store the numbers that the user calls X and Y etc. In fact it stores these variables immediately after the user's program. You can change the place where it starts to store these variables by changing the value of `LOMEM`, but this must be done right at the beginning of the program.

The variable `LOMEM` gives the address of the place in memory above which the computer stores all its variables (except for the resident integer variables `@%` and `A%` to `Z%`).

`LOMEM` is normally set to be the same as `TOP` which is the address of the top of the user program. See the keyword `HIMEM` for more details.

Do not accidentally move `LOMEM` in the middle of a program – the interpreter will lose track of all the variables that you are using.

Examples

```
100 LOMEM=TOP+&100
PRINT LOMEM
PRINT ~LOMEM
```

NOTE: The `~` tells the computer to print the value in hexadecimal.

Description

A pseudo-variable which sets the place in memory above which the **BASIC** interpreter stores dynamic variables – those that are created and destroyed as required. Space is always set aside for the resident variables `@%` to `Z%`. Normally `LOMEM` is set equal to `TOP` which contains the address of the end of the user program.

Moving `LOMEM` in the middle of a program will cause loss of all variables.

Syntax

LOMEM=<numeric>

or

<num-var>=**LOMEM**

Associated keywords

HIMEM, TOP, PAGE

MID$

Purpose

To copy part of one string into another string. For example, if:

```
demo$="DOGMATIC"
```

then the middle part of `demo$`, starting at position four and going on for three letters, i.e. `MID$(demo$,4,3)` would equal `MAT`. In fact `MID$` can be used to copy any part of a string – not just the middle part.

Thus `MID$(demo$,1,3)` would equal `DOG` and `MID$(demo$,5,4)` would be `ATIC`.

This string function is very useful for selecting one word out of a long line. There is a demonstration program under the keyword `GOSUB` and another under the keyword `LEN`.

If the last number is omitted then the function returns with the rest of the string.

Example

```
RESTofLINES=MID$(main$,10)
```

Description

A string function which returns a sub-section of the first argument's string. The second argument gives the starting position and the third argument gives the number of characters to be copied. If the source string is too short then the function returns as many characters as possible from the starting position.

Syntax

<string-var>=`MID$`(<string>,<numeric>[,<numeric>])

Associated keywords

LEFT$, RIGHT$, LEN, INSTR

MOD modulus

Purpose

The function `MOD` gives the remainder after division. When doing division with whole numbers (I emphasise – with ***whole*** numbers) it is sometimes useful to know the remainder. For example 14 divided by 5 leaves a remainder of 4 (14=2*5+4). Similarly:

```
PRINT 14 MOD 5
```

would print `4`. The whole number part of the above division is given by the function `DIV`. Thus:

```
PRINT 14 DIV 5
```

would print `2`.

Notice that the result of both `DIV` and `MOD` is always a whole number.

In fact ***all*** numbers used in the calculation of the function are first converted to integers (using internal truncation) before the computer calculates the result.

Thus:

```
14 DIV 5          gives 2
14.6 DIV 5.1      gives 2
14 MOD 5          gives 4
14.6 MOD 5.1      gives 4
```

The second example (`14.6 DIV 5.1`) is really the same as the first. However:

```
14.6 DIV 4.9      gives 3 and
14.6 MOD 4.9      gives 2
```

are quite different. In effect the computer sees them as:

```
14 DIV 4
14 MOD 4
```

Examples

```
100 LET X=A MOD B
PRINT length MOD 12
```

Description

A binary operation giving the signed remainder of an integer division. **MOD** is defined such that:

```
A MOD B = A-((A DIV B)*B)
```

Syntax

<num-var>=<numeric>**MOD**<numeric>

Associated keywords

DIV

MODE graphics mode

Purpose

This statement is used to select which display MODE the computer is about to use. Changing MODEs clears the screen.

Mode	Graphics	Colour	Text
0	640x256	Two colour display	80x32
1	320x256	Four colour display	40x32
2	160x256	Sixteen colour display	20x32
3		Two colour text only	80x25
4	320x256	Two colour display	40x32
5	160x256	Four colour display	20x32
6		Two colour text only	40x25
7		Teletext display	40x25

MODE 7 uses the Teletext standard display characters. These cannot be changed by the user. Since these characters differ slightly from the standard ASCII set you will find that a number of characters on the screen do not correspond to those printed on the keys. For example a left hand square bracket will be displayed as an arrow.

In MODEs 0 to 6 the character set can be changed by the user. See VDU23 in Chapter 7.

You cannot change MODE inside a procedure or function.

MODEs 128 to 135 are the 'shadow' equivalents of MODEs 0-7. These are only used on the BBC Model B+, and Master Series Microcomputer.

Examples

```
10 MODE 5
MODE 7
```

Description

A statement used to select the display **MODE**, which may not be used in a procedure or function. **MODE** resets the value of **HIMEM**, except when a second processor is in use, or when the computer is operating in the shadow screen mode.

Syntax

MODE<numeric>

Associated keywords

CLS, CLG, HIMEM

MOVE

Purpose

This statement moves the graphics cursor to a particular absolute position without drawing a line. For example to move to a point 100 points across the screen and 300 points up the screen one would say:

```
MOVE 100,300
```

Examples

```
1050 MOVE 100,300
MOVE X,Y
```

Description

To move the graphics cursor to a new position without drawing a line. This statement is identical to **PLOT4**.

Syntax

MOVE<numeric>,<numeric>

Associated keywords

DRAW, MODE, GCOL, PLOT

NEW

Purpose

To 'remove' a program from the computer's memory. In fact the program is still there but the computer has been told to forget about it. If you want to, you can usually recover the old program by typing **OLD**. This only works if you have not entered any part of another program.

NEW is normally used as a command before typing in a new program - to ensure that the computer has forgotten all its previous instructions.

NEW does not clear any of the resident integer variables **A%** to **Z%** or **@%**.

Example

```
NEW
```

Description

A command which resets internal pointers to 'delete' all program statements. The program may be recovered with **OLD** provided no new statements have been entered and no new variables have been created. Since it is a command it cannot form part of a multiple statement line.

Syntax

```
NEW
```

Associated keywords

OLD

NEXT

Purpose

This is used in conjunction with **FOR** to make the computer loop around a set of statements a number of times.

If the loop is opened with (for example):

```
FOR speed=10 TO 100
```

then the **NEXT** statement would normally be in the form:

```
NEXT speed
```

but the word **speed** is optional.

Example

```
340 length=100
350 FOR X=0 TO 640 STEP 2
360 Y=2*length+250
370 DRAW X,Y
380 NEXT
```

Description

A statement delimiting **FOR...NEXT** loops. The control variable (**X** in the last example) is optional.

If a variable is given after **NEXT** then the computer will 'pop' other **FOR...NEXT** loops off the 'stack' until it finds a matching variable. If none is found, an error will be reported.

Syntax

NEXT[<num-var>],[<num-var>],...

Associated keywords

FOR, TO, STEP

NOT

Purpose

This is normally used with an `IF...THEN` statement to reverse the effect of some test.

Example

```
680 IF NOT (A=6 AND B=5) THEN PRINT "WRONG"
```

If `A=6` and `B=5` then the computer will not print `WRONG`.

Description

`NOT` is a high priority unary operator equivalent to unary minus.

Syntax

<num-var>=`NOT`<numeric>

or

<testable condition>=`NOT`(<testable condition>)

OLD

Purpose

To recover a program which has been recently deleted by **NEW** or by pressing the **<BREAK>** key. Programs can only be recovered if no program lines have been entered and if no new variables have been created since the program was deleted. If you get the message **Bad program**, then type **NEW** again.

Typing **NEW** or pressing **<BREAK>** are quite drastic moves. **OLD** will do its best to recover your program but will not always succeed fully. In particular if the first line number in your program is greater than 255 then it will get that one line number wrong. The **<ESCAPE>** key provides a clean method of stopping a program. **<BREAK>** is much more violent and should be avoided.

Example

```
OLD
```

Description

A command which undoes the effect of **NEW**.

Syntax

```
OLD
```

Associated keywords

NEW

ON

Purpose

To alter the order in which **BASIC** executes a program by jumping to one of a selection of lines depending on the value of a particular variable. The word `ON` is used with three other keywords `GOTO`, `GOSUB` and `ERROR`. For example:

```
ON value GOTO 800,920,100 ELSE 7300
ON result GOSUB 8000,8300,120,7600
ON ERROR GOTO 9000
ON ERROR GOSUB 2001
```

First:

```
ON X GOTO 1100,1210,1450,1600,1950
```

If the value `X` is equal to 1 then the program will go to line 1100. If `X=2` then the program will go to line 1210. If `X=3` then line 1450 and so on.

What is it used for? Suppose that you are counting coins put into a machine and you want to offer different things if one, two or three coins are put in. The program which follows illustrates, in outline, how `ON GOTO` will help.

```
 450 REM the variable COINS gives the number
 460 REM of coins inserted
 500 ON COINS GOTO 550,600,650
 550 PRINT "One coin buys a biscuit"
 560 REM gives him a biscuit somehow
 590 GOTO 1000
 600 PRINT "Two coins can buy tea or coffee"
 610 GOTO 1000
 650 PRINT "Three coins can buy a piece of cake"
 660 REM something else in here as well
 690 GOTO 1000
1000 REM all the routines end up here
```

Secondly:

```
ON X GOSUB 2200,2300,2400,2500
```

ON can also be used with **GOSUB** instead of **GOTO**. See **GOSUB** for an explanation of subroutines.

ON X GOSUB provides a neat way of using different subroutines in different situations.

An **ELSE** clause can be included at the end of **ON GOTO** and **ON GOSUB** to trap out of range values without causing an error.

Thirdly:

```
ON ERROR GOTO
ON ERROR OFF
```

If the computer detects an error in your program or in the disc drives or anything else that it can't handle, then it produces an error. In other words it complains and stops. The complaint takes the form of a message on the screen - for example:

```
Too big.
```

Sometimes it is vital that the computer looks after such situations without troubling the user. The statement **ON ERROR GOTO 7000** ensures that if an error occurs the computer does not complain and does not stop. Instead it goes to a section of program at line 7000 (in this case) which has been specially written to get the computer out of the mess it is in. This section of program may have to give the user instructions like **Please enter a smaller number** or it may be able to sort out the problem in some other way.

How well this 'error trapping' works depends on the skill of the programmer in thinking of every possible thing that can go wrong. You will soon re-discover Murphy's Law:

'If anything can go wrong, it will.'

Good error handling is vital in all programs for use by non-specialists - and that means most people!

The statement **ON ERROR OFF** lets the computer deal with errors once again - cancelling the effect of **ON ERROR GOTO**.

Examples

```
40 ON ERROR GOTO 9000
50 ON ERROR PRINT "The computer is confused"
10 ON ERROR GOSUB 2000
```

Description

A statement providing multiple options in changing the order of execution of a program, and error trapping.

Syntax

ON<num-var>**GOTO**<numeric>{,<numeric>}[**ELSE**<statement>]

or

ON<num-var>**GOSUB**<numeric>{,<numeric>}[**ELSE**<statement>]

or

ON ERROR<statement>

or

ON ERROR OFF

Associated keywords

GOTO, GOSUB, ELSE

OPENIN

open file for input to computer from currently selected filing system.

Purpose

To tell the computer that the program wishes to read data (words and numbers). Reading data in is quite a complicated procedure for the computer and it needs advance warning when you wish to do so. The advance warning is given by the **OPENIN** keyword.

One use of this facility is to store names and addresses on file (e.g. the cassette or floppy disc) and to read the file in each time you want to use it. After you have corrected it you can then transfer it back to disc (using **OPENOUT**) where it will be saved for future use. See the chapter on file handling in the manual for the filing system you are using.

A typical example of the use of **OPENIN** is:

```
X=OPENIN("cinemas")
```

This informs the computer that you will shortly want to read data in from a file which is recorded under the name 'cinemas'. The file name is 'cinemas'.

In accepting this instruction the computer allocates a 'channel' to this operation. It is as if it said 'OK that information will be provided on telephone number 6'. It makes **X=6** (or whatever number it decides). In all future operations on that file you must refer to it as channel **X** (channel 6 in this example).

You get the actual data into the computer by using either **BGET#X** or **INPUT#X** as the demonstration program on the next page indicates.

Example

```
230 file=OPENIN("census")
```

Description

A function which attempts to open a file for input or random access. In a floppy or hard disc system, or an Econet environment, if a file already exists with the correct name it will be opened for reading.

The function returns the channel number allocated by the computer's filing system. If the file does not exist then zero is returned.

Syntax

<num-var>=**OPENIN**(<string>)

Associated keywords

OPENOUT, OPENUP, EXT#, PTR#, INPUT#, PRINT#, BGET#, BPUT#, EOF#, CLOSE#

Demonstration program

```
 10 REM to read in the names of 10 cinemas from
 20 REM disc assuming of course that you put
 30 REM them there sometime before!
 50 REM dimension a string array of 10 slots
 60 DIM cine$(10)
 90 REM open the file
100 channel=OPENIN("CINEMA")
110 REM and read in the ten cinema names
120 FOR X=1 TO 10
130 INPUT# channel, cine$(X)
140 NEXT X
150 REM that's the information in
160 REM do whatever you want with it!
```

OPENOUT

open file for output to currently selected filing system

Purpose

This opens a file for output. Before you can record data (rather than programs) you have to open a file. See the chapter on file handling in the manual for the filing system you are using.

`OPENOUT` is used to inform the computer that you wish to store data on the currently selected filing system. The computer allocates a channel to the operation. When working with floppy or hard discs, or over Econet, then if a file already exists with that name it will be deleted. If no file exists then a new one will be created.

Example

```
330 X=OPENOUT("cinemas")
```

Description

A function which returns the channel number allocated to an output file.

If a file of the same name exists then that file will first be deleted. If no file exists then one will be created.

Syntax

<num-var>=`OPENOUT`(<string>)

Associated keywords

OPENIN, OPENUP, PTR#, EXT#, INPUT#, PRINT#, BGET#, BPUT#, EOF#, CLOSE#

OPENUP open a file for update

Purpose

This statement can be used with disc or Econet systems to open a file for update – that is, simultaneous reading and/or writing. With Econet, only one user may open a file for writing at any one time and therefore `OPENUP` should only be used in single user environments.

If a file of the given name exists already then that file will be opened without any changes taking place to the file. If no file of that name exists then `OPENUP` will fail to open the file requested.

`OPENUP` is normally used with random access files on floppy or hard disc or on the Econet filing systems.

Example

```
500 Y%=OPENUP("DATA")
```

Description

A function which returns the channel number allocated to a file opened for both reading and writing. The file must exist before this function can be used.

Syntax

<num-var>=OPENUP(<string>)

Associated keywords

OPENIN, OPENOUT, PTR#, EXT#, INPUT#, PRINT#, BGET#, BPUT#, EOF#, CLOSE#

OPT option

Purpose

This statement determines what output is produced on the screen when assembly language routines are processed by the **BASIC** interpreter. An understanding of the operation of assemblers is required to understand the following.

During assembly two common errors can occur: `Branch out of range` and `Unknown label`.

The latter will occur during pass one for all forward references. It is therefore often desirable to turn off assembler error messages during pass one.

The statement `OPT` is followed by a number in the range 0 to 7, with the following results:

0. Assembler errors suppressed, no listing.
1. Assembler errors suppressed, listing.
2. Assembler errors reported, no listing.
3. Assembler errors reported, listing.

Options 4, 5, 6 and 7 behave exactly as options 0, 1, 2 and 3 except that the code can be placed at a different location from that at which it is intended to execute. With options 4 to 7 the variable P% controls the program counter during assembly, and the variable O% gives the memory location where the code is placed.

The `OPT` statement can only occur inside the square brackets which enclose Assembly Language commands. `OPT` is set to 3 every time the **BASIC** interpreter finds a `[`. Do not confuse it with the operating system command `*OPT`.

Examples

```
200 OPT 1
350 OPT (pass*2+list)
```

Description

An assembler pseudo-operation controlling the output during assembly. `OPT` is followed by an expression as detailed above.

Syntax

OPT<numeric>

Demonstration program

```
 10 oswrch=&FFEE
 20 DIM memory% 100
 30 FOR Z=0 TO 3 STEP 3
 35 P%=memory%
 40 [OPT Z
 50 .start LDA#ASC"!"
 60 LDX#40
 70 .loop JSR oswrch
 80 DEX:BNE loop
 90 RTS
 95 ]:NEXT Z
100 CALL start
110 END
```

OR

Purpose

To enable one condition ***or*** another condition to determine what happens next. The `OR` operator can be used either as a 'logical OR' or as a 'Boolean OR'. See the keyword `AND` for details of logical and Boolean operators.

Example

```
75 IF X=6 OR date>20 THEN PRINT "Good"
```

Description

An operator performing Boolean integer logical **OR** between two numerics.

Syntax

<num-var>=<numeric>OR<numeric>

Associated keywords

AND, EOR, NOT

OSCLI operating system command line interpreter

Purpose

It is very useful in a **BASIC** program to be able to send commands to the operating system. Such commands might include `*FX` commands followed by two numbers. When the program is written you do not always know which numbers are to follow the `*FX` statement. However, you cannot substitute variables such as `X` and `Y` directly after the `*FX` because these variables are not known to the command line interpreter but are only known to the **BASIC** language. Thus the statement:

```
X=5:Y=3:*FX X,Y
```

would be meaningless to the operating system. The statement `OSCLI` provides a neat way of passing variables to the operating system in such cases. `OSCLI` is followed by a string variable which is set to contain the values to be passed to the operating system. Note that numbers must be converted to string form by using the `STR$` function; the above example would work correctly with the following.

```
10 X=5
20 Y=3
30 A$="FX "+STR$X+","+STR$Y
40 OSCLI A$
```

Examples

```
10 FN$="XYZ" : REM FILE NAME
20 START%= &4000 : REM START OF CODE
30 FINISH%= &6000 : REM END OF CODE
40 EXECADD%= &5000 : REM EXECUTION ADDRESS
300 OSCLI "SAVE"+FN$+" "+STR$~(START%)+
" "+STR$~(FINISH%)+" "+STR$~(EXECADD%)
```

Note that no `*` is needed in the string.

Description

A statement which passes its string argument to the operating system command line interpreter.

Syntax

OSCLI<string>

Associated keywords

STR$, CHR$

PAGE

Purpose

PAGE is a variable which gives the address in memory where **BASIC** has stored (or will store) the user's program. This is usually automatically set to be the lowest available address in the computer's Random Access Memory but can be changed by the user.

PAGE can be used to enable the computer to store two different programs at the same time in different areas of memory. Use with care.

Examples

```
PRINT PAGE
10 PAGE=&5000
20 PRINT ~PAGE
235 PAGE=TOP+1000
```

Description

A pseudo-variable giving the address used by the interpreter for the start of the user program. The least significant byte of **PAGE** is always set to zero by the computer. In other words user programs always start on a 'page' boundary where one page is 100 bytes hex (256 bytes decimal).

Syntax

PAGE=<numeric>

or

<num-var>=**PAGE**

Associated keywords

TOP, LOMEM, HIMEM

PI

Purpose

PI has the value 3.14159265. It is used in the example to calculate the area of a circle radius R.

Examples

```
100 AREA=PI*R^2
PRINT PI
```

Description

PI=3.14159265

Syntax

<num-var>=PI

PLOT

Purpose

`PLOT` is the multi-purpose point, line and triangle drawing statement in **BASIC**.

The first number which follows the keyword `PLOT` tells the computer what kind of point, line or triangle it is going to draw. The two following numbers give the X and Y coordinates to be used in plotting the point or drawing the line or triangle.

`PLOT K,X,Y` plots to the point at `X,Y` in a manner determined by the value of `K`. The effect of each value of `K` will be:

0. Move relative to last point.
1. Draw line relative in the current graphics foreground colour.
2. Draw line relative in the logical inverse colour.
3. Draw line relative in the current graphics background colour.
4. Move to absolute position.
5. Draw line absolute in the current graphics foreground colour.
6. Draw line absolute in the logical inverse colour.
7. Draw line absolute in the current graphics background colour.

Higher values of `K` have other effects which are related to the effects given by the values 0 to 7.

8- 15 As 0-7 but with the last point in the line omitted.
16- 23 As 0-7 but with a dotted line.
24- 31 As 0-7 but with a dotted line and without the last point on the line.
32- 63 Reserved.
64- 71 As 0-7 but only a single point is plotted.
72- 79 As 0-7 but to draw a horizontal line to the left and right of the point until a colour other than the current background colour is reached.
80- 87 As 0-7 but plot and fill a triangle.

When filling solid triangles with colour the computer fills the triangle between the coordinates given and the last two points visited.

88- 95 As 0-7 but to draw a horizontal line to the right until reaching the current background colour.

96- 255 Reserved for future expansions.

See Chapter 7 on VDU drivers for an alternative interpretation of the numbers given above.

Suppose that in the above example, `PLOT K,X,Y,` the value of `X` was 50 and the value of `Y` was 80 then 'draw line relative' would mean draw a line to the point on the screen 50 places to the right of the origin and 80 places up from the origin.

Logical inverse colour

In two colour `MODE`s the logical inverse colour of logical colour 0 is logical colour 1.

In four colour `MODE`s the following apply:

Logical	Inverse
0	3
1	2
2	1
3	0

In the 16 colour `MODE` logical colour 0 becomes 15, logical colour 1 becomes 14 and so on.

When drawing lines the computer draws a line from the last point X,Y position given.

Normally the origin is set at the bottom left of the screen, but its position may be moved to any point by using the `VDU29` statement. See Chapter 7 for more information.

The graphics screen is 1280 points (0-1279) wide and 1024 (0-1023) points high.

The most commonly used `PLOT` statements are `PLOT 4` and `PLOT 5`, so these two have been given duplicate keywords; `MOVE` and `DRAW`.

To print a string at a specific place on the screen use the **TAB(X,Y)** statement. As an alternative one can join the graphics and text cursor together with the statement **VDU 5** so that the computer prints text at the graphics cursor position. Once that has been done then the graphics cursor can be moved with **MOVE, DRAW** and **PLOT** statements.

Examples

```
100 PLOT 3,X,Y
PLOT 6,100,220
```

Description

A statement controlling the generation of points, lines and triangles on the screen.

Syntax

PLOT<numeric>,<numeric>,<numeric>

Associated keywords

MODE, CLG, MOVE, DRAW, POINT, VDU, GCOL

POINT

Purpose

To find out the colour of a certain position on the screen. Suppose that you are playing a game involving moving a car around a race track. On the race track are pools of green oil. To find out if the place where your car is about to move to has oil on it (so that the car will skid) you need to be able to find out if the screen is coloured green at that point.

The number returned is the logical colour of the screen at the graphics point specified. If the selected point is off the screen then the number returned will be -1. There must not be a space between the word **POINT** and the opening parenthesis.

Examples

```
1340 colour=POINT(X,Y)
100 IF POINT(X,Y)=2 THEN PRINT "SKID!!"
```

Description

A function returning a number representing the colour on the screen at the specified coordinates. If the point is off the screen then the function returns -1.

Syntax

<num-var>=**POINT**(<numeric>,<numeric>)

Associated keywords

PLOT, DRAW, MOVE, GCOL

POS position

Purpose

This function finds out how far across the screen the flashing cursor is. The left hand side of the screen is position 0 and the right hand side is position 19, 39 or 79 depending on the `MODE` that has been selected.

Examples

```
1005 X=POS
320 distance=POS
```

Description

A function returning the horizontal position of the cursor in the current text window.

Syntax

<num-var>=`POS`

Associated keywords

COUNT, TAB, VPOS

Demonstration program

To print spaces on the screen up to a certain horizontal position – for example to align columns.

```
100 column=12
110 REPEAT PRINT" ";
120 UNTIL POS=column
```

PRINT

Purpose

This does not print anything on paper. It does, however, print words and numbers on the screen.

Anything enclosed in inverted commas (" ") will be printed exactly as it is.

Things not enclosed in inverted commas will be assumed to be variable names and the contents of the variable will be printed out. The exact layout of the numbers and figures on the screen will depend on the punctuation used in the **PRINT** statement.

The items following the word **PRINT** are referred to as the 'print list'.

The screen display is divided into vertical strips (or fields) which are (initially) ten characters wide.

A comma after an item in the print list will cause enough spaces to be printed to ensure that the next item will be printed in the next field.

A semi-colon after an item in the print list will cause the next item to be printed on the same line and immediately following the previous item.

If the print list does not end with a semi-colon then the next **PRINT** statement will print its output on a new line.

PRINT by itself leaves a blank line. A new line can be forced at any stage in the print list by inserting an apostrophe.

The table below gives examples as they would appear, except that commas have been inserted where spaces would be to aid counting.

Example	Print position 12345678901234567890
PRINT 1,2	1 2
PRINT 10,200	10 200
PRINT;10;100	10200
PRINT	
PRINT "Answer";A	Answer42
PRINT "Answer"A	Answer 42
PRINT "Answer",A	Answer 42
PRINT 1/2	0.5
PRINT 1/3	0.333333333
PRINT 3.3'2.25	3.3 2.5

The printer can be turned on at any time by typing <CTRL> <B> or by the statement VDU2 in a program. The output of all PRINT statements will then appear on the printer as well as the screen. <CTRL> <C> turns the printer output off.

Considerable flexibility has been built into the interpreter to enable it to print numbers in several different layouts. There is no need to learn to use these options at first but they will be invaluable when layout is crucial. A more detailed explanation of the advanced features is given below.

It is possible to control the overall field width, the total number of figures printed and the number of decimal places printed.

All these features are set with one variable called @%. In brief, setting:

@%=131594 will give two decimal places
@%=131850 will give three decimal places
@%=&90A will return to the normal output format.

For a detailed understanding of the format it is best to consider @% as a four byte number (e.g. @%=&01020903), each byte controlling one aspect of the print format. The most significant byte will be called B4. It has a value of 01 in the example above. The least significant byte is called B1 and has the value 03 in the example above.

B4 is tested by the function **STR$** to determine the format of strings created by that function. If B4=01 then strings will be formatted paying attention to the setting of **@%** otherwise **@%** will be ignored by **STR$**. Initially B4=00.

B3 selects the **BASIC** format thus:

- 00 General format (G format)
- 01 Exponent format (E format)
- 02 Fixed format (F format)

In G format numbers that are integers will be printed as integers. Numbers in the range 0.1 to 1 will be printed as 0.1 etc. Numbers less than 0.1 will be printed in exponent format.

Exponent format will always print numbers in scientific notation; 100 becomes 1E2, 1000 becomes 1E3 and 1200 becomes 1.2E3.

Fixed format prints numbers with a fixed number of decimal places. If the number cannot be fitted into the selected field width it reverts to G format. The decimal points are aligned vertically which is ideal for scientific and accounting programs.

B2 controls the total number of digits printed in the selected format. If B2 is too large or too small for the **MODE** selected then B2 is taken as 10. The number is rounded to fit in the B2 digit field.

In G format B2 gives the maximum number of digits that can be printed before reverting to E format. Range 1-9.

In E format B2 specifies the total number of digits to be printed before and after the decimal point – but not counting the digits after the E. Another way of looking at it is to say that (B2-1) digits will follow the decimal point. In E format three characters or spaces always follow the final E. Range of B2 in E format is 1 - 10.

In F format B2 specifies the number of digits to follow the decimal point. Range 0-10.

B1 sets the overall print field width and may have any value in the range 0 to 255 which in hexadecimal is &00 to &FF.

For example accounting purposes would often require fixed format two decimal places and ten character field width.

The four bytes of `@%` are built up thus:

`@%=& 00 00 00 00`

B4 - zero `00`

B3 - fixed format `02`

B2 - two decimal places `02`

B1 - character field `0A`

So `@%=&0002020A`, the `&` indicating that the number is in hexadecimal. You can, of course, omit the leading zeros.

Here are some other formats:

Format	(G2)	(G9)	(F2)	(E2)
@%=&	0000020A	0000090A	0002020A	0001020A
100	1E2	100	100.00	1.0E2
10	10	10	10.00	1.0E1
1	1	1	1.00	1.0E0
0.1	0.1	0.1	0.10	1.0E-1
0.01	1E-2	1E-2	0.01	1.0E-2
0.0005	5E-3	5E-3	0.01	5.0E-3
0.001	1E-3	1E-3	0.00	1.0E-3
0	0	0	0.00	0.0E0
-10	-10	-10	-10.00	-1.0E1

Description

A statement causing numeric and string values to be printed on the screen.

Syntax

`PRINT{['][,?;]<string>?<numeric>}['][;]`

Associated keywords

PRINT#, TAB, POS, STR#, WIDTH, INPUT, VDU

PRINT#

Purpose

This records numbers and words on the currently selected filing system. In other words it stores data on a file. Numbers and strings are stored in a special internal format. Before this statement is used the file must have been opened using the **OPENIN**, **OPENOUT** or **OPENUP** statements. See the chapter on file handling in the manual for the filing system you are using.

Example

```
PRINT#file, X,Y,Z,A$,"Monday",33
```

Description

A statement which writes data to files. All values are written in a special internal format:

Integer variables are written as &40 followed by the twos complement representation of the integer in four bytes, most significant byte first.

Real variables are written as &FF followed by four bytes of mantissa and one byte exponent. The mantissa is sent lowest significant bit (LSB) first. 31 bits represent the magnitude of the mantissa and 1 bit the sign. The exponent byte is in twos complement excess 128 form.

String variables are written as &00 followed by a 1 byte 'byte count' followed by the characters in the string in reverse order.

Syntax

PRINT#<num-var>{,<numeric>?<string>}

PROC procedure

Purpose

This is used as the first part of a name to indicate that it refers to a procedure. See the keyword DEF for a fuller description.

Example

```
10 DEF PROChello(X)
20 LOCAL Z
30 FOR Z=0 TO X
40 PRINT "Hello - how about this for BASIC!"
50 NEXT Z
60 ENDPROC
```

Description

A reserved word used at the start of all user declared procedures. There must not be a space between PROC and the rest of the procedure name.

Syntax

DEF PROC<variable-name>[(<string-var>|<num-var>{,<stringvar>|<num-var>}]

PROC<variable-name>[(<string-var>|<num-var>{,<string-var>|<numvar>}]

Associated keywords

DEF, ENDPROC, LOCAL

Demonstration program

```
10 REM Tower of Hanoi problem
20 INPUT "Number of disks",F
30 PROChanoi(F,1,2,3)
40 END
50 DEF PROChanoi(A,B,C,D): IF A=0 ENDPROC
60 PROChanoi(A-1,B,D,C)
70 PRINT "Move disk ";A;" from pile ";B;" to pile ";C
80 PROChanoi (A-1,D,C,B)
90 ENDPROC
```

PTR# pointer

Purpose

This statement is not available on cassette based systems. It selects which item in a long file is to be read or written next. Strings and numbers are stored in a long line one after the other. Each integer number occupies five bytes, each real number occupies six bytes and each string takes up the number of letters in the string plus two. See the keyword **PRINT#** for more details of the file format. The file pointer can be moved up and down the file to point to any selected word or number. Note that you have to keep a careful track of where each word or number starts to use the function. The number immediately following the keyword **PTR#** is the channel number allocated to the file when it was opened. A file must be opened with the **OPENIN** and **OPENOUT** statements. See the chapter on file handling in the manual for the filing system you are using.

Examples

```
PRINT PTR#X
560 PTR#file=PTR#file+80
85 PTR#channel=0
```

Description

A statement and function which allows the programmer to move a pointer to a serial file and thus enables random access.

Syntax

<num-var>=**PTR#**<num-var>

or

PTR#<num-var>=<numeric>

Associated keywords

INPUT#, PRINT#, BGET#, BPUT#, OPENIN, OPENUP, OPENOUT, EXT#, EOF#

RAD radian

Purpose

To convert an angle measured in degrees to radians. A radian equals approximately 57 degrees.

Examples

```
1030 X=RAD(Y)
PRINT RAD(45)
```

Description

A function converting an angular argument given in degrees to radian measure.

Syntax

<num-var>=**RAD**<numeric>

Associated keywords

DEG

READ

Purpose

To enable numbers or words that are required in a program to be made available every time the program is run. It does this by reading numbers or words into numeric or string variables from **DATA** statements in the program. Most often the data is read into an array. See the keyword **DIM** for more information on arrays. See the keyword **DATA** for a more detailed description.

Example

```
100 READ name$(X),A
```

Description

A statement which copies the next item from a data list into the variable or variables which follow the keyword **READ**. The data must contain the correct sequence of string and numeric data for the string and numeric variables to be assigned. In other words numeric data must be supplied if a numeric variable is to be filled.

Syntax

READ<num-var>|<string-var>{,<num-var>|<string-var>}

Associated keywords

DATA, RESTORE

Demonstration Program

```
200 INPUT"How much can you spend",AFFORD
210 PRINT"You can afford the following cars"
220 FOR X=1 TO 10:READ NAME$,PRICE
230 IF PRICE<=AFFORD THEN PRINT NAME$
240 NEXT:END
500 REM VW Based Cars
510 DATA Skoda Oktavia 1.8, 14699
520 DATA etc etc
```

REM remark

Purpose

To enable the program writer to put remarks and comments into the program to help remember what the various parts of the program do. The computer completely ignores anything that appears after a **REM**.

When you first start writing small programs you can get away with having no **REM**s, but as your programs grow in complexity you will find it necessary to have them liberally sprinkled over your program. If you come back to a program six months after you wrote it and find no **REM**s you will have a real job trying to remember how it worked and why you used that variable name etc. Use lots of **REM**s – it will save you hours of time in the long run.

Examples

```
10 REM this revision dated 25th Feb, 2010
100 REM
550 REM data for Ford cars
```

Description

This statement allows comments to be inserted in a program.

Syntax

REM<anything>

RENUMBER

Purpose

When you type in a program you give each instruction a line number. As the program develops you quite often have to insert extra lines between other lines. You might well need to insert 25 lines between line numbers 300 and 310 – difficult!

The `RENUMBER` command will go through your program and renumber it automatically. It recalculates things like `GOTO 220` – which might well become `GOTO 180` etc. However if your program contains the statement `GOTO 100` and there is no line 100 then the `RENUMBER` command will be unable to deal with the problem and will say:

```
Failed at line <new line number>
```

If you renumber a program containing an `ON GOTO` statement which contains a ***calculated*** line number, e.g.:

```
ON X GOTO 120,240,2*R,1000,2000
```

references prior to the calculated line will be renumbered. However it will not recalculate a calculated line number or other line numbers in the same statement – i.e. `2*R,1000` and `2000` in the example given.

The command `RENUMBER` will renumber your program giving the first line the number 10, the second 20 and so on.

The command `RENUMBER 200` will give the first line of your program the number 200, the second will become line 210 etc.

The command `RENUMBER 200,4` would renumber starting with line 200 and then using 204, 208 etc.

`RENUMBER` is a command: it cannot be used in a program, or as part of a multiple statement line.

Examples

```
RENUMBER
RENUMBER 100,20
RENUMBER 6000
```

Description

RENUMBER is a command which renumbers a user's program and will correct most of the cross-references within the program.

Syntax

RENUMBER[<num-const>[,<num-const>]]

REPEAT

Purpose

To make the computer repeat a set of instructions a number of times until some condition is met.

If you jump out of a **REPEAT...UNTIL** loop with a **GOTO** statement (which is bad practice) you must jump back in.

A single **REPEAT** may have more than one **UNTIL**.

Example

```
10 REM print stars for 1 second
20 NOW=TIME
30 REPEAT PRINT
40 UNTIL TIME=NOW+100
```

Description

A statement which is the start of a **REPEAT...UNTIL** loop. These loops always execute once and may be nested up to a depth of 20.

Syntax

REPEAT

Associated keywords

UNTIL

REPORT

Purpose

To get the computer to report in words what the last error was.

Example

```
100 REPORT
```

Description

`REPORT` prints the error message appropriate to the last error condition.

Syntax

```
REPORT
```

Associated keywords

ERR, ERL, ON ERROR

RESTORE

Purpose

Sometimes it is useful to have several sets of data in one program. For example one might want information on Vauxhall cars and on Ferrari cars in the same program. The **RESTORE** statement enables the data pointer to be moved from one set of data to the other.

The word **RESTORE** by itself resets the data pointer to the first set of data in the program.

Examples

```
230 RESTORE
100 RESTORE 6500
RESTORE apointer
```

Description

This statement can be used at any time to reset the data pointer to any selected line number.

Syntax

RESTORE[<numeric>]

Associated keywords

READ, DATA

RETURN

Purpose

The word `RETURN` – not the key marked **<RETURN>** – is used in a program at the end of a subroutine to make the computer return to the place in the program which originally 'called' the subroutine. See `GOSUB` for more details.

There may be more than one `RETURN` statement in a subroutine – but preferably there should be one entry point and one (`RETURN`) exit point.

You should try very hard to avoid leaving a subroutine with `GOTO` – you should always exit with `RETURN`. Why? Well you will soon discover in reasonable sized programs that you can get into an awful tangle and lose track of how a program works if you make the program jump all over the place.

The importance of dividing your programs into clearly defined sections wherever possible, with one entry point and one exit point, cannot be over emphasised.

Examples

```
200 RETURN
300 IF X>4 THEN RETURN
```

Description

A statement which causes the program to branch to the statement after the one which contained the `GOSUB` which called the current subroutine.

Syntax

```
RETURN
```

Associated keywords

GOSUB, ON GOSUB

RIGHT$

Purpose

To copy the right hand part of one string into another string. For example if:

`ABCDE$="HOW ARE YOU"` then
`RIGHT$(ABCDE$,3)` would be "`YOU`" and
`RIGHT$(ABCDE$,7)` would be "`ARE YOU`"

Note that `RIGHT$(ABCDE$,100)` would be "`HOW ARE YOU`" since there are only 11 characters in `HOW ARE YOU`.

Examples

```
A$=RIGHT$(B$,5)
last$=RIGHT$(last$,X)
```

Description

A string function returning a specified number of characters from the right hand end of another string.

Syntax

<string-var>=`RIGHT$`(<string>,<numeric>)

Associated keywords

LEFT$, MID$

RND random

Purpose

To generate a random number.

What exactly this function does is determined by the number which follows the word **RND**.

RND by itself generates a random whole number between -2147483648 and 2147483647.

RND(-X) returns the value -X and resets the random number generator to a number based on X.

RND(0) repeats the last random number given by **RND(1)**.

RND(1) generates a random number between 0 and 0.999999.

RND(X) generates a random whole number between (and possibly including) 1 and X.

The parentheses are compulsory and must immediately follow the word **RND** with no intervening space.

Examples

```
PRINT RND(6)
340 largenumber%=RND
950 PRINT RND(1)
```

Description

A function generating a random number. The range of the number generated depends on the argument (if any).

Syntax

<num-var>=**RND**[(<numeric>)]

RUN

Purpose

To make the computer obey the statements in the program in its memory.

All variables (except the resident integer numeric variables **@%** and **A%** to **Z%**) are first deleted and then the program is executed.

RUN is a statement and programs may therefore execute themselves.

If you want to start a program without clearing all the variables then you can use the statement:

```
GOTO 100
```

or **GOTO** whatever line number you wish to start from, instead of **RUN**.

Examples

```
RUN
9000 RUN
```

Description

RUN is a statement causing the computer to execute the current program.

Syntax

```
RUN
```

Associated keywords

NEW, OLD, LIST, CHAIN

SAVE

Purpose

To save a program that is in the computer's memory onto the currently selected filing system. The program must be given a name – usually called its file name. The file name can have up to seven letters and numbers for the Disc Filing System, or ten letters and numbers for the Cassette Filing System and the Advanced Disc Filing System. The name must start with a letter, and cannot contain spaces or punctuation marks.

Examples

```
SAVE "FRED"
SAVE A$
```

Description

A command which saves the current program area – that is the area between the address given in the variables **PAGE** and **TOP**.

Syntax

SAVE<string>

Associated keywords

LOAD, CHAIN

SGN

Purpose

This determines whether a number is positive, zero or negative. The function returns:

-1	for negative number
0	for zero
+1	for positive number

Examples

```
100 X=SGN(Y)
230 result=SGN(difference)
```

Description

A function returning -1 for an argument which is negative, +1 for a positive argument and zero for an argument equal to zero.

Syntax

<num-var>=**SGN**(<numeric>)

Associated keywords

ABS

SIN sine

Purpose

This calculates the sine of an angle. The angle must be expressed in radians rather than degrees – but you can convert from degrees to radians using the function **RAD**.

Examples

```
120 Y=SIN(RAD(45))
2340 value=SIN(1.56)
```

Description

A function giving the sine of its argument. The argument must be in radians.

Syntax

<num-var>=**SIN**(<numeric>)

Associated keywords

COS, TAN, ACS, ASN, ATN, DEG, RAD

Demonstration program

To draw a sine wave on the screen.

```
10 MODE 4
20 FOR X=0 TO 1280 STEP 4
30 DRAW X,500+500*SIN(X/50)
40 NEXT X
```

SOUND

Purpose

This statement is used to make the computer generate sounds using the internal loudspeaker. The sound generator is capable of making four sounds at once. Each of the four sound channels can generate one note. The keyword **SOUND** must be followed by four numbers which specify:

- Which sound channel is to be used.
- The loudness of the note (or the envelope number).
- The pitch of the note.
- How long the note is to last.

For example:

```
SOUND 1,-15,52,20
```

will play a note on sound channel 1, with a loudness of -15 (maximum volume). A pitch value of 52 gives middle C and a duration of 20 will make the note last for 1 second.

```
SOUND C,A,P,D
```

The channel number (C) can be 0, 1, 2 or 3. Channel 0 is a special channel that can produce various noises, whereas channels 1, 2 and 3 are used to produce single notes. Other values of C produce special effects which are explained further on.

The amplitude or loudness (A) can have any whole number value between -15 and 4. Values -15 to 0 produce notes of fixed loudness throughout the whole note. A value of -15 is the loudest, -7 is half volume and 0 produces silence. Values of 1 to 4 enable the amplitude to be controlled while the note is playing. When you play a note on the piano the sound gradually fades away. Effects like this are selected by using one of the four user defined envelopes which are selected by setting A to be 1, 2, 3 or 4. Envelopes are described in more detail under the keyword **ENVELOPE**.

The pitch (P) is used to set the pitch or frequency of the note. The pitch can have any value between 0 and 255. The note A above middle C is selected with a value of 88.

The table below shows the value of P needed to produce a particular note. You will see that to go up an octave P is increased by 48 and to go up a perfect 5th P must be increased by 28.

	Octave number						
Note	1	2	3	4	5	6	
B	0	48	96	144	192	240	
C	4	*52	100	148	196	244	*middle C
C#	8	56	104	152	200	248	
D	12	60	108	156	204	252	
D#	16	64	112	160	208		
E	20	68	116	164	212		
F	24	72	120	168	216		
F#	28	76	124	172	220		
G	32	80	128	176	224		
G#	36	84	132	180	228		
A	40	88	136	184	232		
A#	44	92	140	188	236		

Increasing the value of P by one will increase the note produced by a quarter of a semi-tone.

To play the chord of C major which consists of the notes C, E and G for two seconds you could enter:

```
100 SOUND 1,-15,52,40
110 SOUND 2,-15,68,40
120 SOUND 3,-15,80,40
```

However to play a number of notes in succession you would enter:

```
100 SOUND 1,-15,96,10
110 SOUND 1,-15,104,10
120 SOUND 1,-15,88,10
130 SOUND 1,-15,40,10
140 SOUND 1,-15,68,20
```

which plays a well-known film theme.

The duration (D) can have any value between -1 and 254. Values in the range 0 to 254 give a note duration of that number of twentieths of a second. Thus if D=40 the note will last for two seconds. Setting D=-1 means that the note will continue to sound until you actually take steps to stop it. You can either press the **<ESCAPE>** key or stop it by sending another note, to the same channel, which has 'flush control' set to 1 – see later in this section.

As was mentioned earlier, channel number 0 produces 'noises' rather than notes and the value of P in the statement:

```
SOUND 0,A,P,D
```

has a different effect from that described for channels 1, 2 and 3. Here is a summary of the effects of different values of P on the noise channel:

P	Effect
0.	High frequency periodic noise.
1.	Medium frequency periodic noise.
2.	Low frequency periodic noise.
3.	Periodic noise of frequency determined by the pitch setting of channel 1.
4.	High frequency 'white' noise.
5.	Medium frequency 'white' noise.
6.	Low frequency 'white' noise.
7.	White noise of frequency determined by the pitch setting of channel 1.

Values of P between 0 and 3 produce a rasping, harsh note. With P set to 4 the noise is like that produced by a radio when it is not tuned to a station – a sort of 'shssh' effect. P=6 sounds like the interference found on bad telephone calls. When P is set to 3 or 7 then the frequency of the noise is controlled by the pitch setting of

sound channel number 1. If the pitch of channel 1 is changed while channel 0 is generating noise then the pitch of the noise will also change. The program below generates a noise on channel 0 and varies the pitch of the noise by changing the pitch of channel 1. Notice that the amplitude of channel 1 is very low (-1) so you will hardly hear it – but you will hear the noise on channel 0.

```
100 SOUND 0,-15,7,150
110 FOR P= 100 TO 250
120 SOUND 1,-1,P,1
130 NEXT P
```

Notice that we have not yet described how sounds can be affected by a superimposed envelope. An envelope can affect both the pitch and amplitude of a note as it is playing. Thus the statement:

```
SOUND 1,-15,255,255
```

merely plays a continuous loud note, but:

```
ENVELOPE 1,1,-26,-36,-45,255,255,255,127,0,0,-127,126,0
SOUND 1,1,255,255
```

produces a complex sound controlled largely by the envelope. See the keyword **ENVELOPE** for more details.

As mentioned briefly at the start of the description of the **SOUND** statement, the channel number, C, can be given values other than 0,1,2 and 3. You do not need to understand exactly why the following works to use it!

For C you can write a four figure hexadecimal number to achieve certain effects – for example:

```
SOUND &1213,-15,52,40
```

The first parameter in the above example has the value **&1213**. The ampersand (**&**) indicates to the computer that the number is to be treated as a hexadecimal number. The four figures which follow the ampersand each control one feature. In this new expanded form the **SOUND** statement looks like:

```
SOUND &HSFC,A,P,D
```

and the functions H, S, F and C will be explained in turn. In essence these numbers enable you to synchronise notes so that you can play chords effectively.

The first number (H) can have the value 0 or 1. If H=1 then instead of playing a new note on the selected channel, the previous note on that channel is allowed to continue. If a note were gently dying away then it might be abruptly terminated when its time was up. Setting H=1 allows the note to continue instead of playing a new note. If H=1 then the note defined by the rest of the **SOUND** statement is ignored.

The second number (S) is used to synchronise the playing of a number of notes. If S=0 then the notes are played as soon as the last note on the selected channel has completed. (There is a slight simplification here; 'completed' means 'has reached the start of the release phase'.) The user is referred to the keyword **ENVELOPE** for relevant detail.

A non-zero value of S indicates to the computer that this note is not to be played until you have a corresponding note on another channel ready to be played. A value of S=1 implies that there is one other note in the group. S=2 implies two other notes (i.e. a total of three). If a note was sent to channel 1 with S set to 1 then it would not be played until a note was ready on another channel which also had S set to 1. For example:

```
110 SOUND &101,-15,50,200
110 SOUND 2,-15,200,100
120 SOUND &102,-15,100,200
```

When this program is run the note at line 100 will not play until channel 2 is free. Line 110 sounds a note immediately on channel 2 – and for five seconds (duration 100). When the note has completed then both the notes from lines 100 and 120 will sound together.

The third number (F) can have the value 0 or 1. If it is set to 1 then the **SOUND** statement in which it occurs flushes (throws away) any other notes waiting in the queue for a particular channel. It also stops whatever note is being generated on

that channel at present. The SOUND statement in which F=1 then plays its note. Setting F behaves like an 'over-ride'. For example:

```
20 SOUND 2,-15,200,100
25 FOR X=1 TO 500:NEXT X
30 SOUND &12,-15,100,200
```

In the above situation line 20 will start a sound on channel 2 but this will be stopped almost immediately by line 30 which will generate a lower and longer note on channel 2. Line 25 just gives a short delay.

Setting F=1 provides an easy way of stopping an everlasting note! Thus SOUND&13,0,0,1 stops the current note on channel 3 and instead plays one at zero loudness and of minimum length. This will stop channel 3 immediately.

The last number (C) is the channel number described earlier.

Description

The sound generator has four separately controlled synthesis channels. Each can sound at one of 16 amplitudes, including 'off'. The audio output is the sum of the channel outputs. Channels 1-3 each generate a square wave with programmable frequency. Channel 0 can produce noise (unpitched sound of pseudo-random structure) or a pulse waveform. The frequency of the pulsewave or period of the noise can be set to one of the three fixed options, or to the frequency of channel 1.

The **BASIC** program generates each sound by initiating one or more 'requests' each of which may take the form of a musical note or a single effect and is directed to a specific channel. If the destination channel is idle when a request requires it, the sound starts playing immediately. If a previous request is still being handled the new one is placed on a queue, where it waits until the current event is over (or past a critical stage – see ENVELOPE). If the queue is full, the program waits. Separate queues are provided for the four channels, each of which can hold up to four requests, not counting the one currently being executed. The program can look at the state of any queue and flush any queue, but cannot find out or alter the state of the current event, except for flushing the whole queue.

The SOUND keyword is followed by four parameters, the first of which consists of four hexadecimal digits. Thus:

```
SOUND &HSFC,A,P,D
```

Parameter	Range	Function
H	0 or 1	Continuation
S	0 to 3	Synchronisation
F	0 or 1	Flush
C	0 to 3	Channel number
A	-15 to 4	Amplitude or envelope number
P	0 to 255	Pitch
D	1 to 255	Duration

The 'H' parameter allows the previous event on that channel to continue and if this is 1 the amplitude and pitch parameters of SOUND have no effect. Because the dummy note is queued in the normal way, it can be used to ensure that the release segment of sound, which occurs after the duration is over and would otherwise be truncated by the next sounding event on the same channel, is allowed to complete.

The 'S' parameter allows requests to be queued separately and then executed at the same instant, for chords and multiple voice effects. The value initially determines the number of other channels that must receive requests with the same value of S before the group will play. For example, each note of a three note chord would be generated by a SOUND with the value of 2 for S. The system will read the value of S from the first one and then wait for two more requests with 2 as the value of S before playing the complete chord. Single requests use 0 for S so that they play as soon as they reach the end of the channel queue.

The parameter 'F' will normally be zero, causing the request to be queued. If it is 1, the channel queue will be flushed first, so the request will sound immediately.

The parameter 'C' determines the number of the sound channel to be used.

The 'A' parameter controls the amplitude of the sound and can be used in two ways. Positive values up to 4 select the envelope (1 to 4) to be used. If the RS423 is unused then envelope numbers up to 16 may be defined and used. Zero and negative integers up to -15 directly set the amplitude of the sound, which is then

fixed at this value for the duration of the note. -15 corresponds to the loudest, and 0 is 'off'.

The 'P' parameter determines the pitch of the note. It can take values from 0 to 255.

The 'D' parameter determines the total duration of sounds whose amplitude is determined explicitly by a negative or zero value of the A parameter. The duration is given in twentieths of a second. If an envelope has been selected, by a positive value of A, then the duration D determines the total of the attack, decay and sustain periods - but not of the release phase.

Syntax

SOUND<numeric>,<numeric>,<numeric>,<numeric>

Associated keywords

ENVELOPE, ADVAL

SPC space

Purpose

This statement is used to print multiple spaces on the screen. It can only be used as part of **PRINT** or **INPUT** statements. The number in parenthesis gives the number of spaces to be printed.

Examples

```
120 PRINT "Name";SPC(6);"Age";SPC(10);"Hours"
4030 INPUT SPC(10),"Value", V
```

Description

A statement printing a number of spaces on the screen. Up to 255 spaces may be printed.

Syntax

PRINT SPC(<numeric>)

or

INPUT SPC(<numeric>)

Associated keywords

TAB, PRINT, INPUT

SQR square root

Purpose

This statement is used to calculate the square root of a number.

Examples

```
10 X=SQR(Y)
300 X=(-B+SQR(B^2-4*A*C))/(2*A)
```

Description

A function returning the square root of its argument. An attempt to calculate the square root of a negative number will produce the error message:

```
-ve root
```

which is error number 21.

Syntax

<num-var>=**SQR**(<numeric>)

Associated keywords

None

STEP

Purpose

This is part of the **FOR...TO...STEP...NEXT** structure.

In the program shown below, **STEP** indicates the amount that the variable **cost** is to be increased each time around the loop. In this case the cost is to increase in steps of five units.

The step may be positive or negative.

STEP is optional. If omitted a step size of +1 is assumed – see the keyword **FOR**.

Example

```
300 FOR X=100 TO 20 STEP -2.3
```

Description

Part of the **FOR...NEXT** construct. **STEP** is optional.

Syntax

FOR<num-var>=<numeric>**TO**<numeric>[**STEP**<numeric>]

Associated keywords

FOR, TO, NEXT

Demonstration program

```
230 FOR cost=100 TO 200 STEP 5
250 production=FNtaken(cost)
260 PRINT production,cost
270 NEXT cost
```

STOP

Purpose

This statement interrupts a program which is running and prints the message

```
STOP at line <line number>
```

on the screen; otherwise the effect is identical to **END**.

STOP may occur as many times as is needed in a program.

Examples

```
2890 STOP
3080 STOP
```

Description

Causes execution of the program to cease and a message to be printed out.

Syntax

```
STOP
```

Associated keywords

END

STR$ string

Purpose

This string function converts a number into the equivalent string representation. Thus `STR$(4.6)` would give `4.6`.

`STR$` is affected by the field width and format constraints imposed by the variable `@%`. The default format is format 0, field width 10 . See the `PRINT` command for more information on `@%`.

The opposite function of converting a string into a number is performed by the functions `EVAL` and `VAL`.

Examples

```
20 A$=STR$(X)
5060 num$=STR$ (size)
10 PRINT STR$~(100)
```

Description

A string function which returns the string form of the numeric argument as it would have been printed.

Syntax

<string-var>=`STR$`(<numeric>)

Associated keywords

VAL, PRINT, EVAL, OSCLI

STRING$

Purpose

This produces a long string consisting of multiple copies of a shorter string. Thus `STRING$(6,"--0")` would be `--0--0--0--0--0--0`. This function is useful for decorative features. It should be used whenever the user needs to generate a long string from identical short strings.

It is very important, to avoid wasting memory space, that strings are set to their maximum length the first time that they are allocated. This can easily be done by using `STRING$`. For example to set `A$` to contain up to 40 characters one could write:

```
A$=STRING$(40," ")
```

`A$` can then be set back to empty using `A$=""` before use.

Examples

```
400 A$=STRING$(x,pattern$)
560 B4$=STRING$(5,"0+")
PRINT STRING$(10,"hello")
```

Description

A string function returning multiple concatenations of a string.

Syntax

<string-var>=`STRING$`(<numeric>,<string>)

TAB tabulation

Purpose

TAB can only be used with the keywords **PRINT** and **INPUT**. There are two versions:

TAB(X) will print spaces up to a certain column position. If the flashing cursor is beyond the required position then the cursor will move to the next line down and space across to the required column.

TAB(X,Y) will move the cursor directly to position X,Y on the screen. Note that once **TAB(X,Y)** has been used on a line **TAB(X)** may not move to the correct position on the line.

The origin (for all text commands) is at the top left of the current text area of the screen.

The left hand column of the screen is column number 0. The right hand column is column 19, 39 or 79 depending on the graphics **MODE** selected.

The top line is line number 0, the bottom line is line number 31 or 24.

If the text scrolling area of the screen is changed then the **TAB** command will still work as outlined above.

Examples

```
340 PRINT TAB(10);name$TAB(30);job$
440 PRINT TAB(20,31);value
230 INPUT TAB(10,20) "How much" cost
875 INPUT TAB(30), "Doctor's name", DOC$
```

Description

TAB with a single argument prints spaces (and a new line if necessary) to reach the specified column.

TAB with two arguments moves the cursor directly to the specified coordinates.

Syntax

PRINT TAB(<numeric>[,<numeric>])

or

INPUT TAB(<numeric>[,<numeric>])

Associated keywords

POS, VPOS, PRINT, INPUT

TAN tangent

Purpose

This mathematical function calculates the tangent of an angle.

The angle must be given in radians but may be converted to radians from degrees using the function **RAD**. A radian is about 57 degrees.

Examples

```
PRINT TAN(RAD(45))
10 Y=TAN(X)
1030 droop=TAN(load)
```

Description

A function returning the tangent of the argument. The argument must be given in radians.

Syntax

<num-var>=**TAN**<numeric>

Associated keywords

COS, SIN, ACS, ATN, DEG, RAD

THEN

Purpose

A keyword used with `IF` to decide on a course of action as the result of some test.

Examples

```
780 IF X=6 THEN PRINT "good" ELSE PRINT "bad"
200 IF A$=B$ THEN PROCgood ELSE PROCbad
```

Description

Optional part of the `IF...THEN...ELSE` structure.

Note that it is not optional if used when the condition assigns to a pseudo variable, e.g.:

```
300 IF X THEN TIME=0
```

Syntax

`IF`<testable condition>[`THEN`]<statement>[`ELSE`<statement>]

Associated keywords

IF, ELSE

TIME

Purpose

This can be used to set or read the internal timer.

The timer counts in one hundredth of a second intervals. It is not a clock providing true time of day readout. However, once set, the internal clock will keep good time. Pressing the <BREAK> key does not reset the clock.

To convert `TIME` to a 24 hour clock use the following routines:

```
1000 SEC=(TIME DIV 100) MOD 60
1010 MIN=(TIME DIV 6000) MOD 60
1020 HR=(TIME DIV 360000) MOD 24
```

Examples

```
205 TIME=(Ho*60+Mi)*60+Se)*100
400 nowtime=TIME
```

Description

A pseudo-variable which sets or reads the lower four bytes of the internal elapsed time clock.

Syntax

`TIME`=<numeric>

or

<num-var>=`TIME`

Demonstration program

```
1070 finishtime=TIME+1000
1080 REPEAT
1090 REM wait for 10 seconds
1100 UNTIL TIME>=finishtime
```

TO

Purpose

Part of the **FOR...TO...STEP...NEXT** statement. The final terminating value of the loop is given after the word **TO**.

Description

Part of the **FOR...NEXT** construct.

Syntax

FOR<num-var>=<numeric>**TO**<numeric>[**STEP**<numeric>]

Associated keywords

FOR, STEP, NEXT

Demonstration program

```
10 MODE 5
20 FOR C=1 TO 3
30 GCOL 3,C
40 FOR X=0 TO 1200 STEP 5*C
50 MOVE 600,750
60 DRAW X,0
70 NEXT X
80 NEXT C
```

TOP

Purpose

The function `TOP` returns the address of the first free memory location after the user's program. The user's program is normally stored from the bottom of the available Random Access Memory upwards.

Thus the length of the user's program in bytes is given by `TOP-PAGE`.

Examples

```
PRINT~(TOP-PAGE):REM length in hex
2340 PRINT TOP
5460 X=TOP
```

Description

A function returning the first free location above the user's program.

Syntax

<num-var>=`TOP`

Associated keywords

PAGE, HIMEM, LOMEM

TRACE

Purpose

TRACE makes the computer print out the line number of each line of the program before execution.

There are three forms of **TRACE**:

TRACE ON	Causes the computer to print line numbers.
TRACE OFF	Turns off the trace facility.
TRACE 6780	Would cause the computer to report only line numbers below 6780.

With well-structured programs which have subroutines at high line numbers this will enable the user to trace through the structure of the program without being bothered with line numbers in procedures, functions and subroutines.

Note that the interpreter does not execute line numbers very often.

```
10 FOR Z=0 TO 100
20 Q=Q*Z:NEXT Z
30 END
```

would print **[10] [20] [30]** but:

```
10 FOR Z=0 TO 100
20 Q=Q*Z
25 NEXT Z
30 END
```

would print **[10] [20] [25] [25] [25] [25] [25]** etc.

(Of course in **MODE 7** the **[** appears as ← and **]** appears as → .)

TRACE is also turned off after an error, or by pressing **<ESCAPE>** or **<BREAK>**.

Examples

```
TRACE ON
TRACE OFF
TRACE X
TRACE 3000
```

Description

TRACE ON causes the interpreter to print executed line numbers when it encounters them.

TRACE X sets a limit on the size of line numbers which may be printed out; only numbers less than X will be printed.

TRACE OFF turns trace mode off.

Syntax

TRACE ON|OFF|<numeric>

TRUE

Purpose

TRUE is represented by the value -1 in this computer.

Examples

```
PRINT TRUE
300 UNTIL result = TRUE
```

Description

A function returning -1.

Syntax

<num-var>=**TRUE**

Associated keywords

FALSE

UNTIL

Purpose

Part of the **REPEAT...UNTIL** construct. See the keyword **REPEAT** for more details.

Example

```
450 UNTIL X<10
```

Description

A program object signifying the end of a **REPEAT...UNTIL** loop.

Syntax

UNTIL<testable condition>

Associated keywords

REPEAT

USR user subroutine

Purpose

The USR function provides the user with a means of calling sections of machine code program which are designed to return one value. When the machine code section is called the computer sets the processor's A, X and Y registers to the least significant bytes of A%, X% and Y%. The carry flag (C) is set to the least significant bit of C%. On return from the machine code section, an integer number is generated from the four registers P, Y, X, A (most significant byte to least significant byte).

Again it must be emphasised that USR returns a result while CALL does not. Therefore you must either assign the result to a variable, e.g.:

```
Registers=USR(&3000)
```

or print the result, e.g.:

```
PRINT USR(&3000)
```

Each individual register may be obtained as follows:

```
10 DIM registers 3
20 !registers = USR(address)
```

After these two lines are executed,

```
Accumulator    =    registers?0
        X      =    registers?1
        Y      =    registers?2
        Flags  =    registers?3
```

Examples

```
1400 R=USR(&3000)
670 result%=USR(plot5)
```

Description

A function allowing machine code to return directly a value for problems which do not require the flexibility of **CALL**.

Syntax

<num-var>=**USR**(<numeric>)

Associated keywords

CALL

VAL value

Purpose

This function takes a string which contains a number and produces the number. In other words it can convert a number represented by a string (e.g.: `A$="+24"`) into the number.

The string must start with a plus (+) or minus (-) sign or a number. If not then the function will return zero.

The opposite function is performed by `STR$`.

Examples

```
450 x=VAL (length$)
1560 date=VAL(DATE$)
```

Description

A function which converts a character string representing a number into numeric form. If the argument is not a signed unary constant then zero will be returned.

Syntax

<num-var>=**VAL**(<string>)

Associated keywords

STR$, EVAL

VDU

Purpose

The statement **VDU** is followed by one or more numbers and the ASCII characters corresponding to these numbers are sent to the screen. The function **CHR$** can generate a single ASCII character from a given number. This character can be added to a string or printed. **VDU** on the other hand is used to generate a sequence of numbers that are then sent to the VDU drivers.

VDU provides an easy way of sending, for example, control characters to the VDU drivers. See Chapter 7 for a detailed list of the VDU control codes.

Two examples will make the purpose of this statement clearer: when defining the text area of the screen four bytes have to follow the **VDU 28** statement. These four bytes represent the left X, bottom Y, right X and top Y coordinates of the text area. In **MODE 4** the range of X is 0-39 and of Y is 0-31. Thus:

```
VDU 28,0,5,39,0
```

would define a six line text window at the top of the screen. If a different **MODE** is selected then the maximum screen width may be either 19, 39 or 79.

The graphics area of the screen, on the other hand, uses coordinates up to 1279 points horizontally. Thus when defining the graphics area, double byte numbers must be sent to the VDU drivers since the largest number that can be sent as a single byte is 255.

```
VDU 24,0;0;1279;830
```

will define a graphics area at the bottom of the screen and 830 points high. Each of the four coordinates is sent as a double byte pair. Note that the graphics origin is bottom left while the text origin is top left and that the graphics screen is always 1280 by 1024 regardless of **MODE**.

VDU is equivalent to **PRINT CHR$;** except that it does not change the value of COUNT.

Examples

`VDU 14` Turn auto-paging mode on.

`VDU 15` Turn auto-paging mode off.

`VDU 2` Turn printer on.

Description

A statement which takes a list of numeric arguments and sends them to the operating system output character routine (OSWRCH). If the argument is followed by a semi-colon then that argument will be sent as two bytes. The least significant byte will be sent first, followed by the most significant byte. This is the order required by the VDU drivers.

Syntax

`VDU` <numeric>{`,` | `;`<numeric>}[`;`]

Associated keywords

CHR$

VPOS vertical position of the cursor

Purpose

VPOS is used to find the vertical position of the text cursor on the screen.

Examples

```
670 V=VPOS
100 PRINT VPOS
```

Description

A function returning the vertical position of the text cursor.

Syntax

<num-var>=VPOS

Associated keywords

POS

WIDTH

Purpose

WIDTH is used to set the overall 'page width' that the computer uses. Initially this is set to zero which the interpreter interprets as 'unlimited width'.

WIDTH n will cause the interpreter to force a new line after n characters have been printed by the **PRINT** statement.

WIDTH also affects all output to the printer.

Examples

```
670 WIDTH 60
WIDTH 35
```

Description

A statement controlling the overall output field width. It is initially set to zero which disables auto new lines.

Syntax

WIDTH<numeric>

Associated keywords

COUNT

6. Error messages

If the computer is unable to proceed for some reason then it will report the fact to you by printing an error message on the screen. The printing of the message can be suppressed by an **ON ERROR statement** – for example:

```
ON ERROR PROCerror
```

ON ERROR may be followed by any statement or multiple statement.

As well as the error message, the computer sets two variables each time an error occurs:

- **ERR** gives the error number.
- **ERL** gives the number of the line in the program where the error occurred.

REPORT is a command to print the last error message. For example:

```
ON ERROR REPORT:PRINT" at line ";ERL:END
```

will give the same response that the computer gives without an **ON ERROR** statement.

The error messages are listed below in alphabetical order together with their error numbers.

Accuracy lost - 23

If you try to calculate trigonometric functions with very large angles you are liable to lose a great deal of accuracy in reducing the angle to the range of plus or minus PI radians. In this case the computer will report Accuracy lost, e.g:

```
PRINT SIN(10000000)
```

Arguments - 31

This error indicates that their are too many or too few arguments for a given function or procedure.

Array - 14

This indicates that the computer thinks that an array is to be accessed but does not know the array in question.

Bad call - 30

This indicates that the use of `PROC` or `FN` to call a defined procedure or function is incorrect.

Bad DIM - 10

Arrays must be dimensioned with a positive number of elements. An error will be produced, for example, by:

```
DIM A(-3)
```

Bad HEX - 28

Hex numbers can only include 0 to 9 and A to F. An attempt to form a hex number with other letters will result in this error, e.g.:

```
PRINT &y
```

Bad key - 251

An attempt has been made to provide a function key definition with a key number greater than 15.

Bad MODE - 25

This indicates an attempt to change mode inside a procedure or function, or to select a mode for which there is insufficient memory.

Bad program - 0

There are a number of occasions on which the computer checks to see if the program that it contains starts and ends in memory. The untrappable and fatal error `Bad program` indicates that the computer could not follow a program through successfully to an end mark in memory. This is caused by a read error or by loading only part of a program or by overwriting part of a program in some way. Unless you are prepared to check the contents of memory a byte at a time there is little that can be done to recover a bad program.

Bad string - 253

A string more than ten characters long has been passed with an operating system command (cassette filing system only).

Block? - 218

This is an error generated by the cassette filing system. It indicates that an unexpected block number was encountered. Rewind the tape a short way and play it again.

Byte - 2

An attempt was made, during an assembly language section, to load a register with a number requiring more than one byte, e.g.:

```
LDA #345
```

Can't match FOR - 33

There is no **FOR** statement corresponding to the **NEXT** statement.

Channel - 222

This error is generated by the current filing system if an attempt is made to use a channel that was not opened.

Data? - 216

This an error generated by the cassette filing system and it means that the computer has found a cyclic redundancy check (CRC) error. The CRC is stored on tape along with other information. Rewind the tape a short way and play it again.

DIM space - 11

An attempt was made to dimension an array for which there was insufficient room.

Division by zero - 18

Division cannot be done, e.g.:

```
PRINT 34/0
```

This error can also be caused by a division within a procedure or function using a **LOCAL** variable which has not been set to a new value. When a variable is declared as **LOCAL** it is set to zero.

$ range - 8

The user may put strings into any place in memory except zero page – that is locations with addresses lass than &100. Thus this is illegal:

```
$40="hello"
```

Eof - 223

This error is generated by the cassette filing system when the end of the file is reached.

Escape - 17

The <ESCAPE> key has been pressed.

Exp range - 24

The function EXP cannot deal with powers greater than 88. Thus the following is illegal:

```
X=EXP(90)
```

Failed at <line number>

When renumbering a program the computer attempts to look after all references made by GOTO and GOSUB statements. Thus the program:

```
133 GOTO 170
170 END
```

would become:

```
10 GOTO 20
20 END
```

when renumbered. However, the computer will not be able to deal with:

```
133 GOTO 140
200 END
```

If the user attempts to renumber this program he or she will get the error message:

```
Failed at line 10
```

and the renumbered program will be:

```
10 GOTO 140
20 END
```

File? - 219

This error indicates that an unexpected file name was encountered by the computer.

FOR variable - 34

The variable in a `FOR...NEXT` loop must be a numeric variable. Thus the following is illegal:

```
FOR 5=3 TO 10
```

Header? - 217

This an error generated by the cassette filing system and it indicates that a header cyclic redundancy check error has occurred. Rewind the tape a short way and play it again.

Index - 3

This indicates an error in specifying an index mode when using the assembler, e.g.:

```
LDA Z,Z
```

Key in use - 250

An attempt has been made to define a function key while another function key is being expanded, e.g.:

```
*KEY 0 *KEY 1 RUN |M |M
```

followed by pressing <kbd>f0</kbd> would produce this error message.

LINE space

The computer has no room left to insert the line in the program.

Log range - 22

An attempt was made to calculate the `LOG` of a negative number or of zero, e.g.:

```
PRINT LOG(-10)
```

Missing , - 5

This error indicates that the computer expected to find a comma in the line, and didn't do so, e.g.:

```
D$=MID$(A$)
```

Missing " - 9

The computer expected to find a double quote, e.g.:

```
LOAD "FRED
```

Missing) - 27

The computer expected to find a closing parenthesis, e.g.:

```
PRINT TAB(10,10
```

Missing # - 45

The computer expected to find a #, e.g.:

```
A=BGET
```

Mistake - 4

This indicates that the computer could not make any sense of the input line.

-ve root - 21

An attempt was made to calculate the square root of a negative number, e.g.:

```
PRINT SQR(-10)
```

This may also occur with `ASN` and `ACS`.

No FN - 7

This indicates that the computer detected the end of a function but had not called a function defenition.

No FOR - 32

A `NEXT` statement was found when no `FOR` statement had been encountered.

No GOSUB - 38

A `RETURN` statement was found when no `GOSUB` statement had been encountered.

No PROC - 13

This indicates that the word `ENDPROC` was found without there being a corresponding `DEF PROC` statement.

No REPEAT - 43

The interpreter found an `UNTIL` statement when no `REPEAT` statement had been encountered.

No room - 0

This untrappable and fatal error indicates that while the computer was running a program it used up all available memory.

No such FN/PROC - 29

If the interpreter meets a name beginning with `FN` (e.g.: `FNfred`) or `PROC` (e.g.: `PROCrob`) it expects to find a corresponding function or procedure definition somewhere. This error indicates that no matching definition was found.

No such line - 41

The computer was told to `GOTO` or `GOSUB` a line number which does not exist.

No such variable - 26

All variables must be assigned to or made `LOCAL`, before they can be accessed in `PRINT` statements or before their values can be assigned to other variables. The initial assignment can simply be, for example, `X=0`.

No TO - 36

A `FOR...NEXT` loop has been set up with the `TO` part missing. A correctly formed line is shown below.

```
FOR X=10 TO 55
```

Not LOCAL - 12

This indicates the appearance of `LOCAL` outside a procedure or function.

ON range - 40

The control variable was either less than 1 or greater than the number of entries in the `ON` list. For example, if `X=3` then the following will fail:

```
ON X GOTO 100,200
```

since there are only two destinations.

ON syntax - 39

The `ON...GOTO` statement was incorrectly formed. For example, the following is illegal:

```
ON X PRINT
```

The word `ON` must be followed by a numeric which must in turn be followed by the word `GOTO` or `GOSUB`.

Out of DATA - 42

An attempt was made to read more items of data than there were in the `DATA` list. The word `RESTORE` can be used to reset the data pointer to the start of the data if required.

Out of range - 1

An attempt was made to branch out of range of the branch instruction in an assembly language program.

Silly

This message will be issued if you attempt to renumber a program or enter `AUTO` mode with a step size of 0 or more than 255, e.g.:

```
AUTO 100,0
```

String to long - 19

The maximum length of a string is 255 characters.

Subscript - 15

This implies that an attempt was made to access an element of an array less than zero or greater than the size of the array. For example, these two lines together will produce this error:

```
100 DIM A(10)
120 A(15)=3
```

Syntax - 220

This error is generated by the cassette filing system and indicates that a syntax error, such as an illegal `*OPT` statement has occurred.

Syntax error - 16

A command was terminated wrongly, for example:

```
LIST PRINT
```

Too big - 20

A number was entered or calculated which was too large for the computer to handle.

Too many FORs - 35

An attempt was made to nest too many `FOR...NEXT` loops. The maximum nesting allowed is ten. This can sometimes be caused by returning to a `FOR` statement without executing a `NEXT` statement, e.g.:

```
10 FOR X=1 TO 6
20 GOTO 10
```

Too many GOSUBs - 37

An attempt was made to nest too many `GOSUB...RETURN` loops. The maximum nesting allowed is 26. This can sometimes be caused by returning to a `GOSUB` statement without executing a `RETURN` statement, e.g.:

```
10 PRINT "WRONG"
20 GOSUB 10
```

Too many REPEATS - 44

An attempt was made to nest too many `REPEAT...UNTIL` loops. The maximum nesting allowed is 20. This can sometimes be caused by returning to a `REPEAT` statement without executing an `UNTIL` statement, e.g.:

```
10 REPEAT
20 GOTO 10
```

Type mismatch - 6

This error indicates that a number was expected and a string was offered or vice versa, e.g.:

```
10 A$=X
```

Table of Error numbers

Error number	Error message
1	Out of range
2	Byte
3	Index
4	Mistake
5	Missing ,
6	Type mismatch
7	No FN
8	$ range
9	Missing "
10	Bad DIM
11	Dim space
12	Not LOCAL
13	No PROC
14	Array
15	Subscript
16	Syntax error
17	Escape
18	Division by zero
19	String too long
20	Too big
21	-ve root
22	Log range
23	Accuracy lost
24	Exp range
25	Bad MODE
26	No such variable
27	Missing)
28	Bad HEX
29	No such FN/PROC
30	Bad call
31	Arguments
32	No FOR
33	Can't match FOR
34	FOR variable
35	Too many FORs

36	No TO
37	Too many GOSUBs
38	No GOSUB
39	ON syntax
40	ON range
41	No such line
42	Out of DATA
43	No REPEAT
44	Too many REPEATs
216	Data?
217	Header?
218	Block?
219	File?
220	Syntax
222	Channel
223	Eof
250	Key in use
251	Bad key
253	Bad string
254	Bad command

NOTE: Filing system errors are described in the appropriate filing system user guide.

7. VDU drivers

The statement `VDU X` is equivalent to `PRINT CHR$(X);` and the statement `VDU X,Y,Z` is equivalent to `PRINT CHR$(X);CHR$(Y);CHR$(Z);`.

However the **VDU** statement is used the most when generating ASCII control codes and a detailed description of the effect of each control code is given in this chapter. The control codes are interpreted by part of the Machine Operating System called the VDU driver.

Programmers writing **BASIC** programs will need to refer to this summary of the VDU drivers if they want to use some of the more advanced facilities such as definition of graphics and text windows. Programmers writing other high level languages or machine code programs will also need to refer to this chapter.

The VDU drivers are part of the Machine Operating System (MOS) software. All high level languages (including **BASIC**) use them to print and draw on the screen. Because they are so extensive and easily accessible to programmers it will be easy to ensure that all high level languages and smaller assembly language programs have access to the same graphics facilities. There is no need for the user to write special routines to handle the screen display.

The BBC Microcomputer is designed so that it can be expanded in many ways. All expansions will be compatible with the current Machine Operating System and it is very important that those writing software use the facilities provided. In a 'twin-processor' machine the only access to the screen memory is via the 'Tube' and use of these VDU drivers and other Machine Operating System features will ensure that code will work correctly whether executed in the input/output processor or in the language processor.

The VDU drivers interpret all 32 ASCII control character codes. Many of the ASCII control codes are followed by a number of bytes. The number of bytes which follow depends on the function to be performed. The VDU code table summarises all the codes and gives the number of bytes which follow the ASCII control code.

Decimal	Hex	CTRL	ASCII abbreviation	Bytes extra	Meaning
0	0	@	NUL	0	Does nothing
1	1	A	SOH	1	Send next character to printer only
2	2	B	STX	0	Enable printer
3	3	C	ETX	0	Disable printer
4	4	D	EOT	0	Write text at text cursor
5	5	E	ENQ	0	Write text at graphics cursor
6	6	F	ACK	0	Enable VDU drivers
7	7	G	BEL	0	Make a short beep
8	8	H	BS	0	Backspace cursor one character
9	9	I	HT	0	Forward space cursor one character
10	A	J	LF	0	Move cursor down one line
11	B	K	VT	0	Move cursor up one line
12	C	L	FF	0	Clear text area
13	D	M	CR	0	Move cursor to start of current line
14	E	N	SO	0	Page mode on
15	F	O	SI	0	Page mode off
16	10	P	DLE	0	Clear graphics area
17	11	Q	DC1	1	Define text colour
18	12	R	DC2	2	Define graphics colour
19	13	S	DC3	3	Define logical colour
20	14	T	DC4	4	Restore default logical colours
21	15	U	NAK	0	Disable VDU drivers or delete current line
22	16	V	SYN	1	Select screen mode
23	17	W	ETB	9	Reprogram display character
24	18	X	CAN	8	Define graphics character
25	19	Y	EM	5	`PLOT` k,x,y
26	1A	Z	SUB	0	Restore default windows
27	1B	[	ESC	0	Does nothing
28	1C	\	FS	4	Define text window
29	1D	]	GS	4	Define graphics origin
30	1E	^	RS	0	Home cursor to top left
31	1F	_	US	2	Move text cursor to x,y
127	7F		DEL	0	Backspace and delete

Detailed description

0 - This code is ignored.

1 - This code causes the next character to be sent to the printer only and not to the screen. The printer must already have been enabled with `VDU2`. Many printers use special control characters to change, for example, the size of the printed output. For example the Epson FX-80 requires a code 14 to place it into double width print mode. This could be effected with the statement:

```
VDU1,14
```

or by pressing **<CTRL>** **<A>** and then **<CTRL>** **<N>**. This code also enables the 'printer ignore' character selected by `*FX6` to be sent to the printer.

2 - This code turns the printer on which means that all output to the screen will also be sent to the printer. In a program the statement `VDU2` should be used, but the same effect can be obtained by typing **<CTRL>** **<B>**.

3 - This code turns the printer off. No further output will be sent to the printer after the statement `VDU3` or after typing **<CTRL>** **<C>**.

4 - This code causes text to be written at the text cursor, i.e. in the normal fashion. A `MODE` change selects `VDU4`, normal operation.

5 - This code causes text to be written where the graphics cursor is. The position of the text cursor is unaffected. Normally the text cursor is controlled with statements such as:

```
PRINT TAB(5,10);
```

and the graphics cursor is controlled with statements like:

```
MOVE700,450
```

Once the statement `VDU5` has been given only one cursor is active (the graphics cursor). This enables text characters to be placed at any position on the screen. There are a number of other effects: text characters overwrite what is already on

the screen so that characters can be superimposed; text and graphics can only be written in the graphics window and the colours used for both text and graphics are the graphics colours. In addition the page no longer scrolls up when at the bottom of the page. Note however that `POS` and `VPOS` still give you the position of the text cursor.

6 - `VDU6` is a complementary code to `VDU21`. `VDU21` stops any further characters being printed on the screen and `VDU6` re-enables screen output. A typical use for this facility would be to prevent a password appearing on the screen as it is being typed in.

7 - This code, which can be entered in a program as `VDU7` or directly from the keyboard as **<CTRL>** **<G>**, causes the computer to make a short 'beep'. This code is not normally passed to the printer.

8 - This code (`VDU8` or **<CTRL>** **<H>**) moves the text cursor one space to the left. If the cursor was at the start of a line then it will be moved to the end of the previous line. It does not delete characters – unlike `VDU127`.

9 - This code (`VDU9` or **<CTRL>** **<I>** or **<TAB>**) moves the cursor forward one character position.

10 - This statement (`VDU10` or **<CTRL>** **<J>**) will move the cursor down one line. If the cursor is already on the bottom line then the whole display will normally be moved up one line.

11 - This code (`VDU11` or **<CTRL>** **<K>**) moves the text cursor up one line. If the cursor is at the top of the screen then the whole display will move down a line.

12 - This code clears the screen – or at least the text area of the screen. The screen is cleared to the text background colour which is normally black. The **BASIC** statement `CLS` has exactly the same effect as `VDU12`, or **<CTRL>** **<L>**. This code also moves the text cursor to the top left of the text window.

13 - This code is produced by the **<RETURN>** key. However, its effect on the screen display if issued as a `VDU13` or `PRINT CHR$(13)`, is to move the text

cursor to the left hand edge of the current text line (but within the current text window, of course).

14 - This code makes the screen display wait at the bottom of each page. It is mainly used when listing long programs to prevent the listing going past so fast that it is impossible to read. The computer will wait until a <SHIFT> key is pressed before continuing. This mode is called 'paged mode'. Paged mode is turned on with <CTRL> <N> and off with <CTRL> <O>. When the computer is waiting at the bottom of a page both the shift lock and caps lock lights will be illuminated.

15 - This code causes the computer to leave paged mode. See the previous entry (14) for more details.

16 - This code (`VDU16` or <CTRL> <P>) clears the graphics area of the screen to the graphics background colour. The **BASIC** statement `CLG` has exactly the same effect. The graphics background colour starts off as black but may have been changed with the `GCOL` statement. `VDU16` does not move the graphics cursor – it just clears the graphics area of the screen.

17 - `VDU17` is used to change the text foreground and background colours. In **BASIC** the statement `COLOUR` is used for an identical purpose. `VDU17` is followed by one number which determines the new colour. See the **BASIC** keyword `COLOUR` for more details.

18 - This code allows the definition of the graphics foreground and background colours. It also specifies how the colour is to be placed on the screen. The colour can be plotted directly, ANDed, ORed or Exclusive-ORed with the colour already there, or the colour there can be inverted. In **BASIC** this is called `GCOL`.

The first byte specifies the mode of action as follows:

0. Plot the colour specified.
1. OR the specified colour with that already there.
2. AND the specified colour with that already there.
3. Exclusive-OR the specified colour with that already there.
4. Invert the colour already there.

The second byte defines the logical colour to be used in future. If the byte is greater than 127 then it defines the graphics background colour (modulo the number of colours available). If the byte is less than 128 then it defines the graphics foreground colour (modulo the number of colours available).

19 - This code is used to select the actual colour that is to be displayed for each logical colour. The statements `COLOUR` (and `GCOL`) are used to select the logical colour that is to be text (and graphics) in the immediate future. However the actual colour can be redefined with `VDU19`. For example:

```
MODE 5
COLOUR 1
```

will print all text in colour 1 which is red by default. However the addition of `VDU 19,1,4,0,0,0` or `VDU 19,1,4;0;` will set logical colour 1 to actual colour 4 (blue). The three zeros after the actual colour in the `VDU19` statement are for future expansion.

In `MODE 5` there are four colours (0, 1, 2 and 3). An attempt to set colour 4 will in fact set colour 0 so the statement:

`VDU 19,4,4,0,0,0` or `VDU 19,4,4;0;`

is equivalent to:

`VDU 19,0,4,0,0,0` or `VDU 19,0,4;0;`

We say that logical colours are reduced modulo the number of colours available in any particular `MODE`.

20 - This code (`VDU20` or **<CTRL>** **<T>**) resets text and graphics foreground logical colours to their default values and also programs default logical to actual colour relationships. The default values are:

Two colour MODEs
0=Black
1=White

Four colour MODEs

0=Black
1=Red
2=Yellow
3=White

16 colour MODE

0=Black
1=Red
2=Green
3=Yellow
4=Blue
5=Magenta
6=Cyan
7=White
8=Flashing black/white
9=Flashing red/cyan
10=Flashing green/magenta
11=Flashing yellow/blue
12=Flashing blue/yellow
13=Flashing magenta/green
14=Flashing cyan/red
15=Flashing white/black

21 - This code behaves in two different ways. If entered at the keyboard (as **<CTRL>** **<U>**) it can be used to delete the whole of the current line. It is used instead of pressing the **<DELETE>** key many times. If the code is generated from within a program by either `VDU21` or `PRINT CHR$(21);` it has the effect of stopping all further graphics or text output to the screen. The VDU is said to be disabled. It can be enabled with `VDU6`.

22 - This VDU code is used to change `MODE`. It is followed by one number which is the new `MODE`. Thus `VDU22,7` is exactly equivalent to `MODE 7` (except that it does not change `HIMEM`).

23 - This code is used to reprogram displayed characters. The ASCII code assigns code numbers for each displayed letter and number. The normal range of displayed

characters includes all upper and lower case letters, numbers and punctuation marks as well as some special symbols. These characters occupy ASCII codes 32 to 126. If the user wishes to define his or her own characters or shapes then ASCII codes 224 to 255 are left available for this purpose. In fact you can redefine any character that is displayed, but extra memory must be set aside if this is done.

ASCII codes 0 to 31 are interpreted as VDU control codes – and this chapter is explaining the exact function of each. The full ASCII set consists of all the VDU control codes, all the normal printable characters and a user defined set of characters.

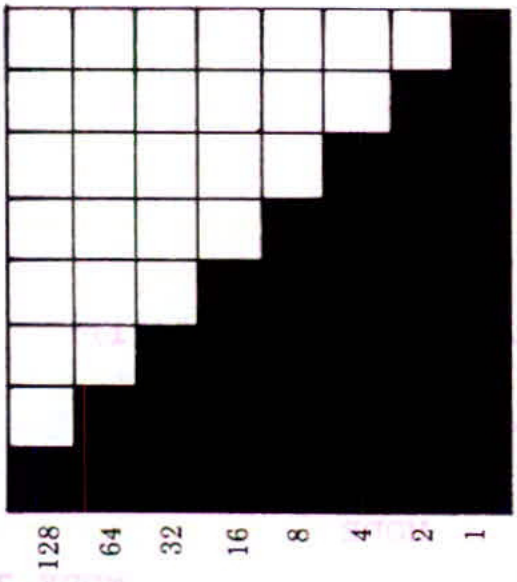
For example if the user wishes to define ASCII code 240 to be a small triangle then the following statement would have to be executed:

Character to be redefined

```
VDU 23,240,1,3,7,15,31,63,127,255
```

Redefine character

Eight numbers giving the contents of each row of dots that makes up the desired character



= 1
2 + 1 = 3
4 + 2 + 1 = 7
8 + 4 + 2 + 1 = 15
16 + 8 . . . = 31
= 63
= 127
= 255

NOTE: You cannot define your own characters in **MODE 7**.

As explained above the user may define any ASCII code in the range 224 to 255. To display the resultant shape on the screen the user can type:

```
PRINT CHR$(240)
```
or

```
VDU 240
```

In the unlikely event of the user wishing to define more than the 32 characters mentioned above (ASCII 224 to 255) it will be necessary to allocate more RAM for the purpose.

`VDU23,1` can be used to turn the flashing cursor off:

```
VDU 23,1,0;0;0;0;
```

will turn the cursor off and

```
VDU 23,1,1;0;0;0;
```

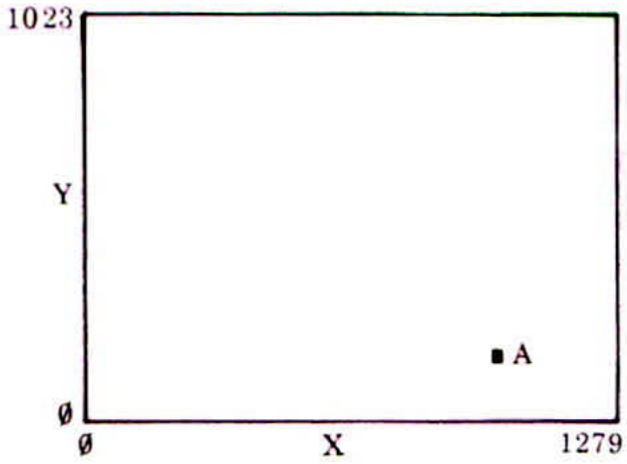
will turn it on again.

A third use of `VDU23` is to permit the advanced programmer to alter the contents of the 6845 CRTC circuit. If the user wishes to place value `X` in register `R` this can be done with the command:

```
VDU 23,0,R,X,0,0,0,0,0,0
```

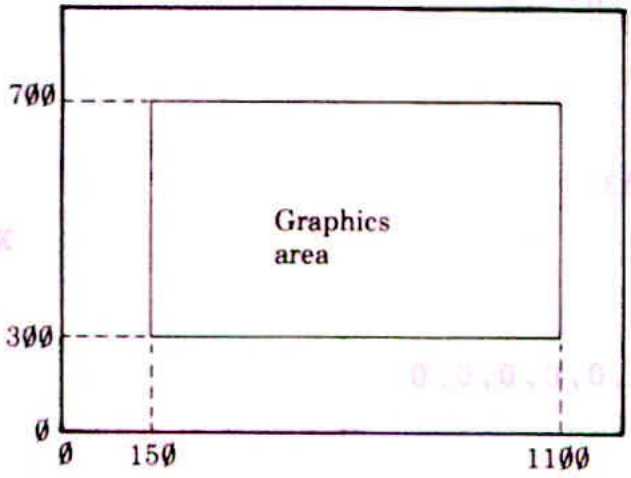
The user is cautioned not to do this unless he or she understands how to program the 6845. Note however that when writing to register 7 (vertical synchronisation position) or register 8 (interlace) of the 6845, any offset that has been set up with the `*TV` statement will be used to adjust the value sent to R7. See Chapter 23 and Appendix F for more details on the 6845 registers.

24 - This code enables the user to define the graphics window – that is, the area of the screen inside which graphics can be drawn with the **DRAW** and **PLOT** statements. The graphics screen is addressed with the following coordinates:



Thus the coordinates of A would be approximately 1000,200.

When defining a graphics window four coordinates must be given; the left, bottom, right and top edges of the graphics area. Suppose that we wish to confine all graphics to the area shown below:



The left hand edge of the graphics area has an X value of (about) 150. The bottom of the area has a Y value of 300. The right hand side has X=1100 and the top has Y=700. The full statement to set this area is:

```
VDU 24,150;300;1100;700;
```

Notice that the edges must be given in the order left X, bottom Y, right X, top Y and that when defining graphics windows the numbers must be followed by a semi-colon.

For those who wish to know why trailing semi-colons are used the reason is as follows: X and Y graphics coordinates have to be sent to the VDU software as two bytes since the values may well be greater than 255. The semi-colon punctuation in the `VDU` statement sends the number as a two byte pair with low byte first followed by the high byte.

25 - This VDU code is identical to the **BASIC** `PLOT` statement. Only those writing machine code graphics will need to use it. `VDU25` is followed by five bytes. The first gives the value of K referred to in the explanation of `PLOT` in the **BASIC** keywords chapter. The next two bytes give the X coordinate and the last two bytes give the Y coordinate. Refer to the entry for `VDU24` for an explanation of the semi-colon syntax used.

For example:

```
VDU 25,4,100;500;
```

would move to absolute position 100,500.

The above is completely equivalent to:

```
VDU 25,4,100,0,244,1
```

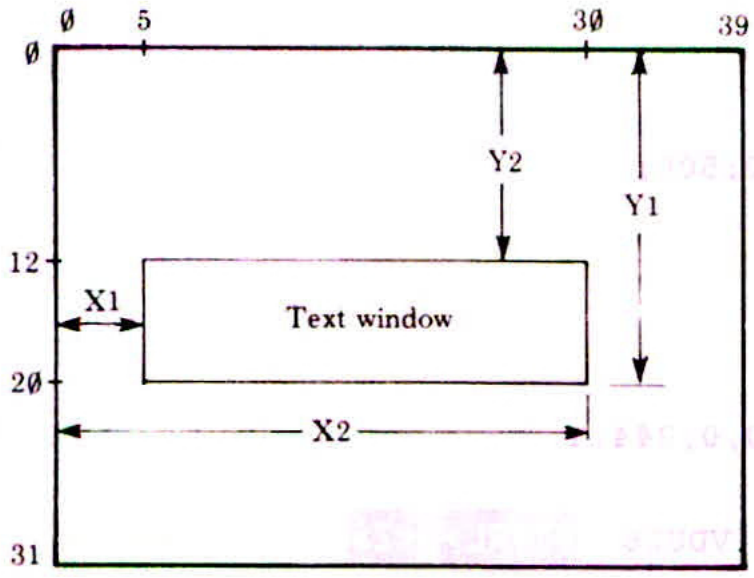
26 - The code `VDU26` (**<CTRL>** **<Z>**) returns both the graphics and text windows to their initial values where they occupy the whole screen. This code repositions the text cursor at the top left of the screen, the graphics cursor at the bottom left and sets the graphics origin to the bottom left of the screen. In this state it is possible to write text and to draw graphics anywhere on the screen.

27 - This code does nothing.

28 - This code (VDU28) is used to set a text window. Initially it is possible to write text anywhere on the screen but establishing a text window enables the user to restrict all future text to a specific area of the screen. The format of the statement is:

```
VDU 28,leftX,bottomY,rightX,topY
```

Where: `leftX` sets the left hand edge of the window
`bottomY` sets the bottom edge
`rightX` sets the right hand edge
`topY` sets the top edge

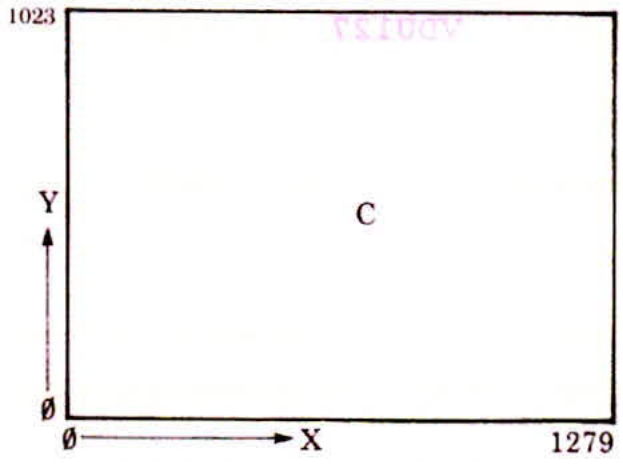


For the example shown the statement would be:

```
VDU 28,5,20,30,12
```

Note that the units are character positions and the maximum values will depend on the `MODE` in use. The example above refers to `MODE 1` and `MODE 4`. In `MODE 2` and `MODE 5` the maximum values would be 19 for X and 24 for Y since these `MODE`s have only 20 characters per line.

29 - This code is used to move the graphics origin. The statement **VDU29** is followed by two numbers giving the X and Y coordinates of the new origin. The graphics screen is addressed as shown below:



To move the origin to the centre of the screen the statement:

```
VDU 29,640;400;
```

should be executed. Note that the X and Y values should be followed by semicolons. See the entry for **VDU24** if you require an explanation of the trailing semi-colons. Note also that the graphics cursor is not affected by **VDU29**.

30 - This code (**VDU30** or **<CTRL>** **<^>**) moves the text cursor to the top left of the text area.

31 - The code **VDU31** enables the text cursor to be moved to any character position on the screen. The statement **VDU31** is followed by two numbers which give the X and Y coordinates of the desired position.

To move the text cursor to the centre of the screen in **MODE 7** one would execute the statement:

```
VDU 31,20,10
```

Note that the maximum values of X and Y depend on the **MODE** selected and that both X and Y are measured from the edges of the current text window not the edges of the screen.

32-126 - These codes generate the full set of letters and numbers in the ASCII set. See the ASCII codes in the Appendices.

127 - This code moves the text cursor back one character and deletes the character at that position. `VDU127` has exactly the same effect as the **<DELETE>** key.

128-223 - These characters are normally undefined and will produce random shapes - see below.

224-255 - These characters may be defined by the user using the statement `VDU23`. It is thus possible to have 32 user defined shapes such as

♣ `VDU 23,224,8,28,28,107,127,107,8,28`
♦ `VDU 23,225,8,28,62,127,62,28,8,0`
♥ `VDU 23,226,54,127,127,127,62,28,8,0`
♠ `VDU 23,227,8,28,62,127,127,127,28,62`

NOTE: You can use the `*FX20` command which will then allow you to define characters 128 to 159 rather than 224 to 255. This has the advantage that you will then be able to use the new characters easily by holding down the SHIFT key while pressing one of the user definable (red) keys. See `*FX20` in Chapter 14 for more details.

8. Cassette files

This chapter summarises the facilities available for file handling using a cassette recorder. If you are using an alternative filing system, such as a floppy or hard disc, Compact Flash, USB, or Econet, then please see the chapter on file handling in the manual for the filing system you are using.

Cassette motor control

Some cassette recorders have a 'remote' or 'automatic' motor control socket. This can be used with a switch on the microphone to start and stop the tape. If your recorder is of this type then the computer will be able to start and stop the tape automatically at the start and end of each **BASIC** program or section of recorded data.

If your cassette recorder does not have motor control then you will have to start and stop the tape manually. A light is provided on the keyboard to tell you when the tape should be running. This is labelled 'cassette motor'. When it is on, the tape should be running.

NOTE: If you have a machine refurbished and supplied by RetroClinic, the cassette motor light may have been modified to show the Compact Flash hard drive activity.

The description which follows assumes that you have automatic or remote motor control. If you don't then you'll have to start and stop the tape manually.

Recording levels

Many cassette recorders employ automatic record level. Recorders of this type do not have any 'record level' controls. If your recorder does not have automatic record level then you will have to set the record level yourself. Set the control so that the recording level indicator is slightly below the '0dB' level or the red mark.

Playback volume and tone

It is important that the playback volume is set correctly. You will need to experiment to find the correct level for your machine. The tone control should normally be set to 'maximum' or 'high'.

Keeping an index of programs

You will be able to record a large number of **BASIC** programs on a single cassette. However it is vital that the programs do not overlap on the tape. If they do then you will lose one of them. Beware of recording on the blank leader tape – always wind it on a little first.

If you forget what is on a tape then you can always use the command:

```
*CAT
```

to obtain a 'catalogue' of the tape. When you give the command `*CAT` (and press the PLAY button on the recorder) the tape will play through, and the computer will print a catalogue of all the programs onto the screen. The catalogue gives the program name, the number of blocks (rather like pages in a book) used to record the program and lastly the length of the program (the number of letters in the book). It also checks that the recording is readable and reports any errors. As the catalogue is building up on the screen you will often see something like this:

```
SKETCH 02
```

This indicates that the computer has found a file called `SKETCH` and that it is currently checking block 2 of that file. The block number is given in hexadecimal not decimal numbers. Press **<ESCAPE>** at the end of the tape to get back control of the computer.

Saving a BASIC program

A program that you have typed into the computer's memory can be saved onto cassette tape in the following way:

1. Insert the cassette into the recorder.
2. Type `SAVE"PROG"` and press **<RETURN>**.

`PROG` is just an example of a file name; file names are explained later in this chapter.

3. The message:

```
RECORD then RETURN
```

will appear. Now use the fast forward and reverse buttons to position the tape at the correct place.

4. Press the RECORD and PLAY buttons on the cassette recorder.
5. Press the **<RETURN>** key on the computer to let it know that everything is now ready.
6. The computer will then record your program.
7. The tape will automatically stop when the computer has finished recording your program.

You can always abandon this process by pressing **<ESCAPE>**.

Saving a section of memory

This will not be needed by most people writing **BASIC** programs. It is most often used to record sections of machine code programs. The process is very similar to that employed to record a **BASIC** program.

1. Insert the cassette in the recorder.
2. Type:

```
*SAVE PROG SSSS FFFF EEEE RRRR <RETURN>
```

3. Continue as for saving a **BASIC** program, above.

`SSSS` represents the start address of the data, in hexadecimal (hex).

`FFFF` represents the end address of the data plus one, in hex. As an option the format `+LLLL` can be used in this position. The plus sign is followed by the length of the data, in hex.

`EEEE` represents the (hex) execution address of the data. If the program is reloaded into the computer using the command:

```
*RUN PROG
```

then once loaded the computer will jump to the specified execution address. The execution address is optional and if it is omitted the execution address will be assumed to be equal to the start address.

`RRRR` represents the (hex) reload address. This is optional, but if used the file will reload (using `*LOAD`, see below) at address `RRRR`. If `RRRR` is omitted then the file will reload at address `SSSS`.

Two examples may make the syntax clearer:

```
*SAVE patch 6000 6200
*SAVE match 4C00+0CE9 2A10 2000
```

Loading a BASIC program

A **BASIC** program saved on cassette tape can be loaded into the computer's memory in the following way:

1. Insert the cassette in the recorder.
2. Type `LOAD "PROG"` **<RETURN>**
3. The message:

```
Searching
```

will appear.

4. Now use the fast forward and reverse buttons to position the tape at the correct place.
5. Press the PLAY button on the cassette recorder.
6. The computer will give you the message:

```
Loading
```

when it finds the correct program. It will then load it into its memory.

7. The tape will automatically stop when the computer has finished loading.

When loading a program the usual catalogue-type display will appear. The message:

```
Loading
```

will appear when the correct file is found. If the load should fail for any reason an error message will appear.

Loading a machine code program

This will not be needed by most people using **BASIC** programs. It is used to load special purpose programs. The process is identical to that used to load a **BASIC** program except that the command is:

```
*LOAD PROG AAAA
```

`AAAA` represents the absolute load address. It is optional but, if included, will force the program to load at the specified address. It therefore over-rides the reload address given when the program was saved. The program will load but not run; control will return to **BASIC**.

Two examples may make the syntax clearer:

```
*LOAD patch
100 MODE 7: *LOAD match 7E80
```

Loading and running a BASIC program

The statement `CHAIN` allows a **BASIC** program to `LOAD` and `RUN` another program. It is particularly useful when there is a sequence of related programs.

The command is used in exactly the same way as `LOAD` but with the word `CHAIN` substituted for the word `LOAD`.

1. Insert the cassette in the recorder.
2. Type `CHAIN "PROG"` **<RETURN>**
3. The message:

```
Searching
```

will appear.

4. Now use the fast forward and reverse buttons to position the tape at the correct place.
5. Press the PLAY button on the cassette recorder.
6. The computer will give you the message:

```
Loading
```

when it finds the correct program. It will then load it into its memory.

7. The tape will automatically stop when the computer has finished loading and the computer will automatically run the program.

Loading and running a machine code program

A machine code program (not a **BASIC** program) can be loaded and run by using the statement:

```
*RUN PROG
```

Using a cassette file to provide keyboard input

It is possible to get the computer to accept input from a cassette file instead of from the keyboard. In this case the cassette file would contain a set of commands, or answers to questions which a **BASIC** program would need. The command to force the computer to accept input from a file called `edit` would be:

```
*EXEC edit
```

File `edit` above is known as an EXEC file. EXEC files can contain **BASIC** commands or operating system commands (or both). A comment line can be included in an EXEC file using a **BASIC** `REM` statement or by beginning the comment with `*|`.

To create a suitable cassette file you will need to use the **BASIC** statement `BPUT#` and not `PRINT#`, since the latter stores things in internal format. The command `*SPOOL` also creates suitable files.

Reading cassette data files

Data, as well as programs, may be recorded on cassette tape. This facility enables the user to keep records of names and addresses (for example) on tape for later use. Since the cassette tape can be started and stopped by the computer it also enables it to record results from experiments over many hours.

If the user wishes to read a data file then he or she must first open the file for input. In the process of opening a file the computer will allocate a channel number to the operation. If we wished to read in a list of names recorded on a data file called `NAMES` then the following statement would get the channel number into the variable `X`.

```
100 X=OPENIN("NAMES")
```

Once a file has been opened for input data can be read in from the tape. This can be done in two ways: a chunk at a time (for example a whole name) or a single letter at a time.

Data is read in a chunk at a time by using the `INPUT#X,A$` statement. This will read the first entry into `A$`.

To read a single character in use the function `A=BGET#(X)`.

Testing for end of file

While reading data in it is useful to test to see if the end of the file has been reached. This is done with the function `EOF#`. For example:

```
100 DIM B$(20)
110 Y=1
120 X=OPENIN("NAMES")
130 REPEAT
140 INPUT#X,A$
150 PRINT A$
160 B$(Y)=A$
165 Y=Y+1
170 UNTIL EOF#X
180 CLOSE#X
190 END
```

The program above reads up to 20 names off tape, prints them on the screen and then stores them in an array in memory. Of course if there were more than 20 names on file then the program would fail because the array can only hold 20 entries.

Storing data on tape

The data in the last example was read into an array in the computer. It could then be edited and the corrected version could be rerecorded on cassette.

The process of recording the data on the cassette consists of three steps: open the file for output, write out the data and then close the file. The example program records 20 entries back to tape from the array in memory.

```
200 X=OPENOUT("NEWNAMES")
210 FOR Y=1 TO 20
220 PRINT#X,A$(Y)
230 NEXT Y
240 CLOSE#X
250 END
```

Note that line 200 will make the computer issue the message:

```
RECORD then RETURN
```

as it did when saving a **BASIC** program.

The `CLOSE#` statement will record any remaining data and then stop the recorder automatically.

Recording single characters on tape

Single characters (bytes) can be placed on tape using the command **BPUT#**. The following program stores the alphabet on tape. Note that the letter A has an ASCII value of 65 and the letter Z has an ASCII value of 90.

```
100 X=OPENOUT("ALPHABET")
110 FOR D=65 TO 90
120 BPUT#X,D
130 NEXT D
140 CLOSE#X
150 END
```

File names

File names on cassettes can be up to ten characters long and can include any character except space.

Responses to errors

If an error reading the cassette occurs during any of the following operations:

```
SAVE
LOAD
CHAIN
*SAVE
*LOAD
*RUN
```

an error message and then the message:

```
Rewind tape
```

will be printed on the screen. The user must then rewind the tape a short way and play it again. It is not usually necessary to replay the whole program, but only far enough to load the blocks causing errors.

If an error is detected during:

```
BGET#
BPUT#
```

INPUT#
PRINT#
*EXEC
*SPOOL

the tape will stop and an error will be generated. This error can be trapped by **BASIC** in the usual way.

The user can escape at any time by pressing the <ESCAPE> key.

The error numbers generated by the cassette file system are as follows:

216 Data cyclic redundancy check (CRC) error.
217 Header CRC error.
218 An unexpected block number was encountered.
219 An unexpected file name was encountered.
220 Syntax error – for example an illegal ***OPT** statement.
222 An attempt was made to use a channel which was not opened.
223 End of file reached.

Changing responses to errors

If the user wishes to change the way the computer behaves when it detects an error during a cassette file operation (e.g. LOAD or SAVE) then this can be done with the *OPT statement.

*OPT by itself resets error handling and the 'message switch' to their default values. Default values are given later.

*OPT1,X is used to set the message switch which controls the detail of the message.

X=0 Implies no messages are issued.
X=1 Gives short messages.
X=3 Gives detailed information including load and execution addresses.

*OPT2,X is used to control the action that the computer takes when it detects an error during a cassette file operation.

X=0 Lets the computer ignore all errors though messages may be given.
X=1 The computer prompts the user to retry by rewinding the cassette.
X=2 The computer aborts the operation by generating an error.

***OPT3,X** is used to set the inter-block gap used when recording data using **PRINT#** and **BPUT#.** The gap on **SAVE** is fixed. The value of X determines the gap in 1/10 seconds. If X is set to greater than 127 then the gap is set to the default value.

The default values for the ***OPT** command are:

For **LOAD, SAVE, CHAIN, *SAVE, *LOAD, *RUN**:

***OPT1,1**	Short messages.
***OPT2,1**	Prompt for retry.
***OPT3,6**	0.6 second inter-block gap.

For **BGET#, INPUT#, BPUT#, PRINT#**:

***OPT1,0**	No messages.
***OPT2,2**	Abort on error.
***OPT3,25**	2.5 second inter-block gap.

NOTE: The effect of the ***OPT** command is different for each filing system. When writing programs that are to run on cassette and disc the user must be fully aware of these different effects. See the chapter on file handling in the manual for the filing system you are using.

Cassette tape format

The format of each block of data stored on cassette is given here for those who wish to produce tapes on other machines that may be read into the BBC Microcomputer.

- Five seconds of 2400Hz tone.
- One synchronisation byte (&2A).
- File name (one to ten characters).
- One end of file name marker byte (&00).
- Load address of file, four bytes, low byte first.

- Execution address of file, four bytes, low byte first.
- Block number, two bytes, low byte first.
- Data block length, two bytes, low byte first.
- Block flag, one byte.
- Spare, four bytes, currently &00.
- CRC on header, two bytes.
- Data, 0 to 256 bytes.
- CRC on data, two bytes.

Notes:

1. Each data block has its own header as shown above.
2. Load and execution addresses should have the top two bytes set to &FF in 8 bit machines.
3. Bit 7 of the block flag is set on the last block of a file.
4. CRC stands for cyclic redundancy check.

The header CRC acts on all bytes from the file name to the spare bytes inclusive. The data CRC acts on the data only. CRCs are stored high byte first and are calculated as follows. In the following C represents the character and H and L represent the high and low bytes of the CRC.

```
H=C EOR H
FOR X=1 TO 8
T=0
IF (bit 7 of H=1) THEN HL=HL EOR &0810:T=1
HL=(HL*2+T) AND &FFFF
NEXT X
```

The above algorithm is not a **BASIC** program!

Changing filing systems

The previous section dealt in detail with cassette files because the BBC Microcomputer is fitted with all the circuitry to enable cassette recorders to be used to store and to retrieve data.

Other filing systems can, of course, be used with the BBC Microcomputer. However, each of these will require additional hardware which will be accompanied by its own detailed instructions. Consequently, details are not given in this guide, for example, of the disc filing system. What is given below, however, is a list of the commands to enable the user to change from one filing system to another.

***TAPE**	Selects the cassette filing system running at the default speed of 120 characters per second (1200 baud).
***TAPE3**	Selects the cassette filing system running at 30 characters per second (300 baud).
***TAPE12**	Selects the cassette filing system at 1200 baud.
***DISC**	This statement selects the disc filing system so that all future file operations (**LOAD**, **SAVE** etc) work to the floppy disc units.
***ADFS**	Selects the advanced disc filing system, and allows the use of hard discs and Compact Flash hard drives
***NET**	Selects the Econet filing system for all future file operations.
***ROM**	Selects the sideways ROM filing system.
***IEEE**	The IEEE 488 interface filing system.
***TELESOFT**	The Teletext filing system.
***RAM**	The RetroClinic RamFS Filing system, for use with the USB DataCentre expansion.
***HADFS**	The Harston ADFS filing system.

These are just some examples, there are many others.

9. Indirection operators

Indirection operators are not normally available on personal computers. They enable the user to read and write to memory in a far more flexible way than by using PEEK and POKE. They are intended for programmers using the computer's assembler or programmers wishing to create their own data structures. There are three indirection operators:

Name	Purpose	Number of bytes affected
? query	Byte indirection operator	1
! shriek (or pling)	Word indirection operator	4
$ dollar (or string)	String indirection operator	1 to 256

For the sake of illustration let us play with the memory around location &2000 (that is 2000 hex – an appreciation of hex numbers is essential. If you don't understand hex then you will not need to use indirection operators). Let us set the variable `M` to &2000:

```
M=&2000
```

`?M` means the contents of memory location `M`, so to set the contents of &2000 to 40 we write:

```
?M=40
```

and to print the contents of memory location &2000 we write:

```
PRINT ?M
```

Those familiar with other dialects of **BASIC** will realise that:

Y=PEEK(X) becomes `Y=?X` and
POKE X,Y becomes `?X=Y`

The query operator acts on one byte of memory only. The word indirection operator acts on four successive bytes. For example:

```
!M=&12345678
```

would set location:

```
&2000=&78
&2001=&56
&2002=&34
&2003=&12
```

and **PRINT ~!M** (print in hex, pling M) would give:

```
12345678
```

Four bytes are used to store integers so the pling operator can be used to manipulate integer variables.

The last operator in this group is the string indirection operator called dollar, or simply string. Do not confuse **$M** with **M$** as they are quite different. **M$** is the usual string variable. On the other hand **$M** can be used to read or write a string to memory starting at memory location **M**. Remembering that we have set:

M=&2000 then:

$M="ABCDEF" will place the string **ABCDEF** and a carriage return (&0D) in memory from location &2000 on.

Similarly **PRINT $M** will print:

```
ABCDEF
```

And one last twist to the use of these operators. We have seen how query, pling and string can be used as unary operators – that is with only one operand. We have used **M** as the operand in the example above – for example:

```
M=&2000
?M=&45
Y=?M
```

```
PRINT ?M

!M=&8A124603
Y=!M
PRINT !M

$M="HELLO ZAPHOD"
B$=$M
PRINT $M
```

but in addition both query and pling can be used as binary operators provided that the left hand operand is a variable (such as `M9`) and not a constant.

Thus `M?3` means the contents of (memory location M plus 3) – in other words of location &2003.

There is a simple routine to examine the contents of memory for 12 bytes beyond &2000.

```
10 FOR X=0 TO 12
20 PRINT ~M+X, ~M?X
30 NEXT
```

Line 20 reads 'print in hex (M plus X) and in the, next column, in hex, the contents of (M plus X)'. It is easy to write this into one of the user defined function keys and keep it for debugging – like this:

```
*KEY 0 FOR X=0 TO 12: P.~M+X,~M?X:NEXT|M
```

but don't forget that in **MODE 7** it will be displayed as

```
*KEY 0 FOR X=0 TO 12: P.÷M+X,÷M?X:NEXT|M
```

just to complicate matters!

Here are some illustrations of some of the above.

Set up function key 0 and use it to examine memory beyond &2000.

```
>*KEY 0 FOR X=0 TO 12:P.~M+X,~M?X:N.|M
>M=&2000
><F0>
>FOR X=0 TO 12:P.~M+X,~M?X:N.
2000 FF
2001 FF
2002 FF
2003 FF
2004 FF
2005 FF
2006 FF
2007 FF
2008 FF
2009 FF
200A FF
200B FF
200C FF
```

Use the byte indirection operator to change one byte.

```
>M?3=&33
><F0>
>FOR X=0 TO 12:P.~M+X,~M?X:N.
2000 FF
2001 FF
2002 FF
2003 33
2004 FF
2005 FF
2006 FF
2007 FF
2008 FF
2009 FF
200A FF
200B FF
```

```
200C FF
```

Use the word indirection operator to change four bytes.

```
>M!2=&12345678
><F0>
>FOR X=0 TO 12:P.~M+X,~M?X:N.
2000 FF
2001 FF
2002 78
2003 56
2004 34
2005 12
2006 FF
2007 FF
2008 FF
2009 FF
200A FF
200B FF
200C FF
```

Use the string indirection operator to insert a string into a known place in memory.

```
>$M="ABCDEFG"
><F0>
>FOR X=0 TO 12:P.~M+X,~M?X:N.
2000 41
2001 42
2002 43
2003 44
2004 45
2005 46
2006 47
2007 D
2008 FF
2009 FF
200A FF
200B FF
200C FF
```

Note that interesting structures can be generated using `!` and `$`. For example you may need a structure containing a ten character string, a number and a reference to a similar structure. `!` and `$` together can do this. If `M` is the address of the start of the structure then:

`$M`	is the string
`M!11`	is the number
`M!15`	is the address of the next structure

The tools are therefore provided to enable you to manipulate general tree structures and lists very easily in **BASIC**.

10. Operating System commands

The command-line interpreter resident within the operating system will recognise a number of commands and act upon receiving them. These commands are most appropriately often used from the keyboard or in **BASIC** programs using the '*' prefix. Commands may also be passed to the command-line interpreter using the OSCLI call (&FFF7) from machine code.

Any command offered to the command-line interpreter but not recognised as a command resident in the command table is offered to paged ROMs for possible action. If it is not claimed by a paged service ROM it is presented to the currently selected filing system ROM. The filing system may recognise the command as one of its internal file commands or attempt to load and execute a file specified by the command name (the cassette and ROM filing systems excepted), i.e. it is equivalent to `*RUN <command name>`.

Each command name may be abbreviated by using enough letters to identify the command terminated by a full stop. The minimum abbreviations for each command are noted below with the description of each command.

Unlike **BASIC**, the command-line interpreter does not distinguish between upper and lower case characters in the command name (`*cat` has the same effect as `*CAT`).

Where a string or filename is specified as a parameter the text need not be enclosed within paired quotation marks but must be separated from the command name by at least one space.

Several of the operating system commands invoke OSBYTE calls and so their action mimics these calls. Where there is an equivalent OSBYTE call this has been indicated.

A number of commands are filing system dependent. Any command which creates or uses files is described for the ROM and cassette filing systems only.

*|

An operating system command-line with a '|', *string escape character,* as its first non-blank character will be ignored by the operating system. This could be used to put comment lines into a series of operating system commands placed in an EXEC file for example.

*.

This command is directly equivalent to the ***CAT** command.

*/<file name>

This command is treated exactly the same as typing ***RUN <file name>** or ***<file name>**. The file name is offered directly to the filing system and will not be interpreted as a command.

*BASIC (*B.)

While the operating system is independent of any particular language and only serves to provide an interface between languages or utility software and the machine hardware, **BASIC** has been accorded special status. The ***BASIC** command is resident in the operating system command table and it enables the operating system to select a language in paged ROM not possessing a service entry point (for further details about paged ROMs see Chapter 20). The operating system scans the paged ROMs and keeps a record of which paged ROM contains **BASIC** (see OSBYTE &BB/187). If a **BASIC** ROM is not present this command is offered to other paged ROMs.

CAT (.)

This command displays a catalogue of files from the selected filing system. When the cassette or ROM filing systems are selected the name of each file encountered is printed on the screen along with the block number of the last block read. When the last block is reached further information is printed out. If the default messages are selected then the length of the file will be added to the block number:

```
FILENAME 09 0904
```

If extended messages have been selected then the catalogue printout after the final block has been reached will look like this:

```
FILENAME 09 0904 FFFF0E00 FFFF801F
```

This file is a **BASIC** (level 1) program SAVEed from **BASIC** with PAGE=&E00 and the program is &904 bytes long. The fourth field in the catalogue printout is the start address of the file and the file will *LOAD to this address by default. The fifth field is the execution address. When using level 2 **BASIC** the execution address will be &8023. When an attempt is made to *RUN a file the processor jumps (using JSR) to this address. The two most significant bytes of the four-byte fourth and fifth fields are set to the machine high order address (see OSBYTE &82/130).

For details about selecting extended messages see *OPT.

*CODE x,y (*CO.) OSBYTE with A=&88 (136)

This command enables the user to incorporate his own command into the operating system command table. *CODE executes machine code indirected through the user vector (USERV) at locations &200, &201 (low-byte, high-byte). The default contents of the user vector produce the Bad Command message. The machine code at USERV is entered with A=0, X=x and Y=y.

For example:

```
 10 DIM MC% 100
 20 OSASCI=&FFE3
 30 USERV=&200
 40 FOR opt%=0 TO 3 STEP 3
 50 P%=MC%
 60 [
 70 OPT opt%
 80 .write
 90 CMP #0                    \ is this *CODE ?
100 BEQ code                  \ if *CODE call act upon it
110 BRK                       \ anything else, print error message
120 ]
130 ?P%=255:P%=P%+1:          REM error number
140 $P%="*CODE only please"
150 P%=P%+LEN$P%
160 [
170 OPT opt%                  \ reset OPT
180 BRK                       \ op code value=0
```

```
190 .code TXA                   \ transfer contents of X reg. to Acc.
200 JSR OSASCI                  \ print ASCII character
210 RTS                         \ return to BASIC
220 ]
230 NEXT
240 ?USERV=write MOD 256
250 ?(USERV+1)=write DIV 256
```

This example prints out the ASCII character corresponding to the value of the first parameter given to the ***CODE** command. After this program has been run typing in ***CODE 65** or ***FX 136,65** will cause a letter **A** to be printed. The second parameter (stored in Y) if included, is ignored.

See also ***LINE**.

*EXEC <filename> (*E.)

Text files from the currently selected filing system can be used as if they were keyboard input using this command. A typical application might involve the setting up of a user's favourite soft key definitions which are ***EXEC**ed in at the beginning of a programming session.

See also ***SPOOL** and OSBYTE &C6/198.

*FX a,x,y (*F.)

OSBYTE calls may be performed directly from the keyboard using this command. A, X and Y are loaded by the operating system from the command-line parameters. Any OSBYTE call may be made using the ***FX** command but it is not always appropriate to make an OSBYTE call using this direct method e.g. OSBYTE calls that return values in any of the registers. The ***FX** command is a useful way of making those OSBYTE calls which have a direct effect from a **BASIC** program or from the command-line interpreter. For further information on specific ***FX**/OSBYTE calls refer to Chapter 14.

*HELP (*H.)

Typed in on a machine containing only the operating system and **BASIC** ROMs this command causes the version number of the operating system to be printed out i.e.:

```
OS 1.20
```

Each *HELP call is offered to any paged ROMs that are resident and these may be able to respond to further command-line parameters, e.g.:

```
*HELP VIEW
*HELP UTILS
```

For more information about *HELP handling in paged ROMs see Chapter 20 (service call 9).

*KEY n <string> (*K.)

The ten red-topped function keys, the **<BREAK>** key, the **<COPY>** key and the four cursor control keys may be set up using this command. Using *KEY n with n in the range 0 to 9 sets up the function keys. *KEY10 can be used to program the **<BREAK>** key.

Before the remaining programmable keys can be used a *FX4,2 must be performed. This disables cursor editing and enables the following soft keys:

*KEY 11	COPY
*KEY 12	left cursor
*KEY 13	right cursor
*KEY 14	down cursor
*KEY 15	up cursor

Each time a soft key is pressed a soft key character is inserted into the keyboard buffer. The soft key characters may be calculated by adding the soft key number to &80 (128). (A soft break actually places a character of value &CA in the input buffer but this behaves identically to character &8A.)

See OSBYTEs &DD/221 to &E4/228 for more information about the function keys.

Control codes may be introduced into the string by using an 'escape' character, '|'. This acts in a similar way to the <CTRL> key and so 'G' gives a bell sound as the <CTRL> and <G> keys would if pressed simultaneously; |G actually places a character of value 7 into the soft key buffer. The '|' character may be inserted using the sequence '||' and a quotation mark (") may be represented by preceding the quotation mark with the 'escape' character ('|'). The delete character (ASCII &7F/127) may be introduced using the 'escape' character followed by a question mark. Characters of value greater than 127 may be inserted by using the escape sequence '|!'; this sequence will add 128 to the value of the next character in the string, e.g.:

```
|!A     128+65=193
|!|A    128+1=129
```

This method of including non-printable characters into a string may be used in any string which is processed using the operating system routines GSINIT and GSREAD and is not restricted to use of the *KEY command.

*LINE <text> (*LI.)

This command executes machine code at the location pointed to by the contents of the user vector (USERV) at locations &200,&201 (low-byte, high-byte). The command enters this code with the A=1, X=least significant byte of string address and Y=most significant byte of string address. *LINE provides an easy method of incorporating a user function into the operating system command table.

```
 10 DIM MC% 100
 20 OSASCI=&FFE3
 30 USERV=&200
 40 FOR opt%=0 TO 3 STEP3
 50 P%=MC%
 60 [
 70 OPT opt%
 80 .write
 90 CMP #1                    \ is this *LINE?
100 BEQ code                  \ execute machine code if *LINE
110 BRK                       \ otherwise print Out error message
120 ]
130 ?P%=255:P%=P%+1:          REM error number
140 $P%="*LINE only please"
150 P%=P%+LEN$p%
160 [
```

```
170 OPT opt%                    \ reset OPT
180 BRK                         \ op code value 0
190 .code STX &70               \ *LINE code entry point and store
200 STY &71                     \ string address; low-byte,high-byte
210 LDY #0                      \ set up Y register for indexing
220 .loop LDA(&70),Y            \ Post-Indexed Indirect addressing
230 JSR OSASCI                  \ print Out character
240 INY                         \ increment index
250 CMP #&0D                    \ test for end of string
260 BNE loop                    \ if not last character go round again
270 RTS                         \ finished
280 ]
290 NEXT
300 ?USERV=write MOD 256
310 ?(USERV+1)=write DIV 256
```

This example program sets up the user vector to point to some machine code which prints out the string pointed to by X and Y. After this program has been run typing in:

```
*LINE THIS IS SOME TEXT
```

results in `THIS IS SOME TEXT` being printed out.

See also `*CODE`.

*LOAD <filename> <address> (*L.)

A file may be loaded into memory from the selected filing system using the `*LOAD` command. If the load address is not specified in the command line then the file will load at its start address (this is usually the address from which it was saved).

*MOTOR n (*M.) OSBYTE with A=&89 (137)

This command executed with n=0 opens the cassette relay (i.e. switches the motor off) and with n=1 closes the cassette relay (i.e. motor on).

*OPT x,y (*O.) OSBYTE with A=&8B (139)

This command is highly filing-system specific and although the general protocol of the cassette filing system is usually adhered to, other filing systems may interpret this command differently and expand upon it.

For the cassette and ROM filing systems:

`*OPT 0,0`	restore *OPT default values
`*OPT 1,0`	turn off filing system messages
`*OPT 1,1`	turn on filing system messages (non-extended)
`*OPT 1,2`	turn on extended messages
`*OPT 2,0`	errors ignored though messages may be given
`*OPT 2,1`	on error, prompt for re-try
`*OPT 2,2`	on error, abort
`*OPT 3,n`	set interblock gaps to n/10 seconds (only relevant to cassette SAVE operations)

When extended messages have been selected the following information is printed on the screen on completion of the filing system operation:

file name - block no. - file length - start adr. - execution adr.

All numeric values are printed in hexadecimal.

See also `*CAT`.

`*ROM (*RO.)` OSBYTE with A=&8D (141)

The `*ROM` filing system is initialised using this command. The `*ROM` filing system is able to use paged ROMs or serially accessed ROMs associated with the speech processor. These ROMs must contain data in a block format similar to that used in the cassette filing system. With the ROM filing system initialised all other filing systems are disabled.

For further details see Chapter 21 on Filing Systems.

*RUN <file name> (*R.)

This command causes a file to be loaded into memory at its start address and then the microprocessor jumps (using JSR) to the execution address. This is a method of loading and running machine code programs. Any text following the file name is available to pass parameters to the program. Parameter passing is not implemented for the cassette or ROM filing systems (see Chapter 21).

*SAVE <file name> <start addr> <end addr> <exec.addr> <reload addr> (*S.)

The contents of memory may be saved to a file on the currently selected filing system using *SAVE. Only the start address and the end address are mandatory. If omitted the execution address will default to the start address. The reload address allows the start address stored with the file to be different to the actual start address used when saving. The end address may be in the form:

```
<+length>
```

where the second field is preceded by a '+' and the size of memory to be saved is specified in hexadecimal.

*SPOOL <filename> (*SP.)

The *SPOOL command causes all screen output to be repeated into a file. The file is opened by *SPOOL <file name> and closed by repeating this command or by typing *SPOOL alone.

See OSBYTEs &03 and &C7/199 for more information.

*TAPE n (*T.) OSBYTE with A=&8C (140)

*TAPE without any number selects cassette filing system and sets the default baud rate (1200). *TAPE 3 selects tape with 300 baud and *TAPE 12 selects 1200 baud.

*TV x,y (no abbreviation) OSBYTE with A=&90 (144)

The *TV command allows the vertical position of the screen to be altered and interlace to be switched on or off. The first parameter causes the vertical position to be altered; a value of 0 causes no change, a value of 1 would cause the screen to be moved up one line and a value of 255 would cause the screen to be moved down one line. The second parameter should be 0 or 1, a value of 0 causes interlace to be enabled and a value of 1 causes interlace to be switched off. Any change of interlace or screen position will only come into effect at the next mode change and will remain until a further `*TV` command or a hard reset. Interlace cannot be turned off in mode 7.

It is possible to switch off interlace in mode 7 but the character set stored in the SAA 5050 is designed to be used with interlace on. Type in:

```
VDU23,0,8,&90;0;0;0,23,0,9,&09;0;0;0
```

and you will see why the operating system disallows this. See Chapter 23 for more information about programming the 6845 video controller chip.

11. Machine code and the BASIC Assembler

There comes a time in every programmer's life (well, most programmers', anyway) when the constraints of a high level language (e.g. **BASIC**) prevents him from implementing a particular program idea or from utilising some machine facility. At this stage the programmer must often seek recourse to the microprocessor's native language, its machine code.

At the heart of any microcomputer is the microprocessor. This microprocessor is the brain of the computer and provides the computer with all its computing power. The BBC Microcomputer uses a 6502 microprocessor and the brief description of machine code given here applies specifically to the 6502. The microprocessor performs instructions which are contained in memory. Each instruction which the microprocessor understands can be contained within a single byte of memory. Depending on the nature of this instruction the microprocessor may fetch a number of bytes of data from the memory locations following the instruction byte. Having executed this instruction the microprocessor moves on to the byte after the last data byte to get its next instruction. In this way the microprocessor works its way sequentially through a program. These single byte instructions are called *operation codes* (or just opcodes) although they are frequently referred to as machine instructions (or just instructions). The data used by the instructions are called the *operands*. A program using the native machine instructions is called a *machine code* program. An *assembler* is a software package (a language or a program) which enables a programmer to create a machine code program.

Machine code is substantially different from a higher level language such as **BASIC**. The machine code programmer is limited to three registers for temporary storage of data while in **BASIC** he has unlimited use of variables. For more permanent storage in machine code programs, the register values can be copied to bytes of memory. Only very limited arithmetic is available; there are no multiply or divide instructions. There are no automatic loop structures such as `FOR... NEXT` or `REPEAT... UNTIL` and any loops must be explicitly set up by the programmer using conditional branches (these approximate to `IF .. THEN GOTO ..` in

BASIC). The range of instructions available are sufficient to enable extremely complex programs to be written but a lot more effort is required to implement the program. One of the most grave disadvantages of machine code is that very little error checking is made available to the programmer. A well designed assembler will help the programmer, but once the machine code program is running the only error checking is that which is provided within the program itself.

At first glance it may appear that there is little to be gained from writing a machine code program. The principal advantage is that of speed. While assigning a value to a variable in **BASIC** will take about 1 millisecond, in machine code assigning a similar value will only take 10 microseconds, around 100x faster. This is why fast moving arcade games have to be written in machine code. Some of the facilities available on the BBC Microcomputer can only be used when programming in machine code. For example, a user printer driver can only be implemented in machine code.

Of no less importance than the design of the hardware or the choice of microprocessor in the machine is the operating system. This is a large, and highly complex machine code program which governs the machine. The operating system consists of a large number of routines which perform operations such as scanning the keyboard, updating the screen, performing analogue-to-digital conversions (for the joysticks) and controlling the sound generator. All these functions are performed by the operating system and are made available to other machine code programs. A machine code program with which all users will be familiar is **BASIC**. It is this program which provides the user with an easier way of using the microprocessors computing power, and it constantly uses the operating system routines to get input from the keyboard and to reflect that input on the screen. **BASIC** recognises words of text, and when it wants to use a hardware facility it calls a machine code routine within the operating system. The great advantage of this independence of the language program from the direct use of hardware is that the same facilities can be offered to different languages.

Writing a machine code program requires the programmer to place the appropriate values into successive memory locations corresponding to the opcodes and operands. This would be a very tedious business if it had to be done by looking up the opcode values in tables and poking in the values by hand. It is much faster, easier and more efficient to get the computer to do most of the work. A program

which analyses text input representing opcode symbols, converts these to opcode values and inserts these values into memory is called an *assembler*. The text input consists of opcode mnemonics (three-letter words which specify the opcode type) followed by numbers, variable names or expressions which give the values of the operand to be used by the opcode. Like **BASIC** the assembler requires the program to be written in a defined way according to a syntax. The language that an assembler understands is called assembler or assembly language. In the BBC Microcomputer an assembler is available as part of the **BASIC** language and a description of how this assembler can be used is contained in the following section.

The BASIC Assembler

One of the many attractive features of **BBC BASIC** is the incorporation of a mnemonic assembler within the language itself. This provides a powerful environment for the assembler and allows machine code to be easily incorporated within **BASIC** programs. Hybrid **BASIC**/machine code programs may often lead to the use of the best features of each language, the speed of machine code when it is required, coupled with the increased power of **BASIC** when speed is not of paramount importance.

The assembler facilities available to users are dependent on the version of **BASIC** that is resident in the machine. To ascertain which version of **BASIC** is present type `REPORT` following a **<BREAK>**. If the copyright message is dated 1981 then this is 'old **BASIC**' which will henceforth be referred to as Level 1 **BASIC**, and if the message is dated 1982 then this is 'new **BASIC**' which will be referred to as Level 2 **BASIC**.

Below is an example of a simple machine code program written using the **BASIC** assembler.

```
10 OSWRCH=&FFE3
20 DIM MC% 100
25 DIM data &20
30 FOR opt%=0 TO 3 STEP 3
40 P%=MC%
50 [
60 OPT opt%
70 .entry LDX #0            \ set index count (in X reg.) to 0
75 LDA data                 \ load first item in accumulator
80 .loop JSR OSWRCH         \ perform VDU command
```

```
 90 INX                          \ increment index count
100 LDA data,X                   \ load next VDU parameter
110 CPX #&20                     \ has count reached 32 (&20) ?
120 BNE loop                     \ if not then go round again
130 RTS                          \ back to BASIC
140 ]
150 NEXT opt%
160 !data=&04190516
170 data!4=&00C800C8
180 data!8=&00000119
190 data!&C=&01190064
200 data!&10=&000000C8
210 data!&14=&00000119
220 data!&18=&0119FF9C
230 data!&1C=&0000FF38
240 CALL entry
```

This program performs some simple graphics using the **BASIC** VDU method to select the screen **MODE** and perform **PLOT**ting. All the VDU codes are contained within the block of memory labelled 'data'. Using the operator does not make it immediately obvious what is going on. Four bytes are inserted into memory with each operator. The least significant byte being inserted at the address specified. Each subsequent byte is inserted into the next byte of memory. i.e.:

```
!data=&04190416
data!4=&00C800C8
```

will result in an equivalent to:

```
VDU &16,&04,&19,&04,&C8,&00,&C8,&00
```

or, to separate it into its two components:

```
VDU &16,&04
VDU &19,&04,&00C8;&00C8;
```

or:

`VDU 22,4` select **MODE 4**

```
VDU 25,4,200;200;
```

PLOT 4,200,200 - move absolute X,Y

Any program which can be written in **BASIC** may also be implemented in machine code although it is not always sensible to do so.

There now follows a detailed description of using the **BASIC** mnemonic assembler.

The assembler delimiters '[' and ']'.

All the assembler statements should be enclosed within a pair of square brackets. When the **BASIC** program is `RUN`, the assembler statements contained between the square brackets are assembled into machine code. This code is inserted directly into memory at the address specified by P%, and P% is incremented by the number of bytes in each instruction or directive.

Within the assembler delimiters the text of the assembly language program may be written. The assembly language program will consist of a number of assembler statements separated by new lines or colons (as in **BASIC**).

Each assembler statement should consist of an optional label followed by an instruction (this will be a three letter assembler mnemonic or an assembler directive) and an operand (or address). If a label is included it should be separated from the instruction by at least one space. The operand need not be separated from the instruction. Any character following the operand and separated by at least one space from it will be totally ignored by the assembler which will move onto the next colon or line for the next statement. A comment may be placed after the operand field and should be preceded by an backslash (\). Any text following an backslash in an assembly statement will be ignored by the assembler up to the next colon or end-of-line.

NOTE: In level 1 **BASIC** colons cannot be included in expressions. Missing out a colon in a multi-statement line will result in the statement after the intended colon being ignored by the assembler. This error is often difficult to spot in a program which assembles without error but then fails to function as the programmer had anticipated.

During assembly of the example program above the following printout is produced (with PAGE=&1900):

```
>RUN
```

```
1BEA
1BEA
1BEA            OPT opt%
1BEA A2 00      .entry LDX #0          \set index count (in X reg.) to 0
1BEC 00 5A 1C   .loop LDA data,X       \load next VDU parameter
1BEF 20 E3 FF   JSR OSASCI             \perform VDU command
1BF2 E8         INX                    \increment index count
1BF3 EQ 20      CPX #&20               \has count reached 32 (&20)?
1BF5 00 F5      BNE loop               \if not then go round again
1BP7 60         RTS                    \back to BASIC
```

OPT - assembler option selection

OPT is an assembler directive or non-assembling statement which can be included within an assembly program to select a number of different assembler options.

The OPT command should be followed by a number to make the option selection. The assembler options are selected on the state of the least significant 2 or 3 bits of the OPT parameter.

bit 0 if set, assembly listing enabled.
bit 1 if set, assembler errors enabled.
bit 2 if set, assembled code placed in memory at O%
(Implemented in Level 2 **BASIC** only)

In the example program above OPT is set up using the `FOR...NEXT` loop variable, `opt%`. On the first pass of the assembler `OPT 0` is used, listing is suppressed and assembler errors are not enabled. For the second pass an `OPT 3` is used which switches on assembly listing and enables assembler errors. **BASIC** errors will be flagged as normal. The assembler errors which are suppressed are the `Branch out of range` error and the `No such variable` error. These will normally be generated during the first pass when the assembler is resolving forward passes.

Bit 2 allows a program to be assembled into one region of memory while being set up to run at a different address. P%, the program counter (see below) should be set up as usual to provide the source of label values. If bit 2 is set then O% should be set up at the same time as P% to point to the start of memory into which the machine code is to be assembled. This facility is useful for assembling machine code where it is impossible to use the memory in which the program is eventually

going to reside (e.g. Assembling programs which are going to be blown into EPROM for paged ROMs). This option is only available in Level 2 **BASIC**.

Each time the assembler is entered the OPT value is initialised to 3. This means that a second chunk of assembler in the same **BASIC** program must perform its own OPT selection.

The Location Counter P%

When the assembler is creating the machine code program the code produced is placed in memory starting from the address in P% (one of the resident integer variables) unless remote assembly has been selected using OPT.

The programmer must set P% to a meaningful value before the assembly begins. The usual method for short programs is to `DIM`ension a block of memory and to set P% to this value at the beginning of each pass of the assembler (as in the example above). A classic problem is sometimes encountered when a programmer adds more code to a short program which has been allocated space by this method. If the code created overflows the space `DIM`ensioned for it and is over-written by **BASIC**, it will fail to operate as expected when tested; alternatively the code may over-write the **BASIC** dynamic storage and a `No such variable` error will be flagged during the second pass of the assembler.

The assembler updates P% as it is assembling and when it reaches the end of a pass the value of P% represents the address of the first 'free' byte of memory after the machine code program.

Labels

Any **BASIC** numerically assignable item may be used as a label with the assembler (such as a variable or an array element). A label is defined by preceding the variable name with a full stop. The full stop prefix causes the assembler to set up a **BASIC** variable containing the current value of P%. Once set up this variable is available for use by any other part of the assembler or **BASIC** program.

Forward Referencing and Two Pass Assembly

In the construction of a machine code program using the **BASIC** assembler a large number of labels may be generated. It is often the case that one part of the program needs to jump forward over another part of the program. Labels provide a convenient way of marking that point in the program to which the processor is to jump. When assembling the machine code, the assembler works sequentially through the program and in the case of a forward reference the assembler will encounter the reference before the label. In the normal course of events an error will be flagged `No such variable`. In order to resolve forward references, two passes of the assembler are required. The first pass should be performed with error trapping switched off and during this pass all the labels will be initialised. A second pass will provide all the correct values required for forward referencing. During this second pass error trapping should be enabled to pick up any genuine programming mistakes.

The most convenient way of performing the two passes is to use a `FOR...NEXT` loop. The programmer should make sure that P% is reinitialised at the beginning of the second pass. It is often convenient to set up the pseudo-operation `OPT` using the `FOR` loop variable (errors and listing disabled for the first pass, errors enabled and listing as required for the second).

The EQUate Facility in Level 2 BASIC

One of the improvements made to Level 2 **BASIC** was the incorporation of some `EQU` pseudo-operation commands. These allow the incorporation of data by reserving memory within the body of the assembly language program.

The `EQU`ate operations available are:

`EQUB`	equate byte	reserves 1 byte of memory
`EQUW`	equate word	reserves 2 bytes of memory
`EQUD`	equate double	word reserves 4 bytes of memory
`EQUS`	equate string	reserves memory as required

These operations initialise the reserved memory to the values specified by the address field. The address field may contain a string, in double quotes, or string variable for the `EQUS` operation or a number or numeric variable for the other

EQU operations. The assembler will use the least significant part of the value if too large a value is specified.

The example program, written in Level 2 **BASIC**, could have been written with lines 30 and 170 to 240 replaced with:

```
141.data EQUD &04190516
142 EQUD &00C800C8
143 EQUD &00000119
144 EQUD &01190064
145 EQUD &000000C8
146 EQUD &00000119
147 EQUD &O119FF9C
148 EQUD &0000FF38
```

In Level 1 **BASIC** one way to reserve space for data within the body of a machine code program is to leave the assembler using a right-hand square bracket and insert the data using the address contained in P%. P% should then be incremented by the appropriate amount before entering the assembler.

e.g. to incorporate a string into a machine code program:

```
 10 DIM MC% 100
 20 OSRDCH=&FFE0
 30 OSASCI=&FFE3
 40 FOR opt%=0 TO 3 STEP3
 50 P%=MC%
 60 [
 70 OPT opt%
 80 .entry LDY #0              \ zero loop index
 90 .loop LDA string,Y         \ load accumulator with Y?string
100 JSR OSASCI                 \ write the character
110 INY                        \ increment loop index
120 CMP #&0D                   \ is the current character a CR
130 BNE loop                   \ if not get the next character
140 JSR OSRDCH                 \ get character from keyboard
150 CMP #9                     \ is it the TAB key
160 BNE error                  \ if not flag an error
170 RTS                        \ return to BASIC
180 .string
190 ]
200 $P%="Please press the TAB key'
210 P%=P%+LEN($P%)+1
220 [
230 OPT opt%
240 .error BRK                 \ cause an error
```

```
250 ]
260 NEXT opt%
270 ?P%=&FF
280 P%=P%+1
290 $P%="Wrong key pressed"
300 ?(P%+LEN($P%))=0
310 CALL entry
```

This program prompts the user to press the TAB key by printing out a message. If the wrong key is pressed an error is flagged.

Handling errors with BRK

In the example program above the **BRK** instruction is used to generate an error. The **BRK** instruction forces an interrupt which is interpreted by the operating system as an error. As part of the error handling in **BASIC** the programmer can incorporate an error number and an error message into his code to identify the error. The byte in memory following the **BRK** instruction should contain the error number. The error message string should follow the error number and must be terminated by a zero byte.

The following lines set this up:

```
240 .error BRK                  \cause an error
270 ?P%=&FF Error number 255
280 P%=P%+1
290 $P%="Wrong key pressed":  REM Error message
300 ?(P%+LEN($P%))=0:         REM Terminating byte
```

When a **BRK** is encountered in a machine code program called from **BASIC** the error message is printed out together with the line number from which the machine code was called. Typing **REPORT** or printing **ERR** will reproduce the message and error number as with any **BASIC** error.

The user can provide his own BRK handling routine which may be useful when using machine code away from the **BASIC** environment.

Entering machine code from BASIC - CALL and USR

Machine code routines can be entered from a **BASIC** program using either the `CALL` statement or the `USR` function. On entry to the machine code program using these instructions, the accumulator, the X register, the Y register and the carry flag are set to the least significant bytes (or bit) of the resident integer variables A%, X%, Y% and C%. A number of parameters may be passed to the machine code routine if the `CALL` statement is used, the addresses and data types of these parameters being available to the machine code in a parameter block at location &600. The `USR` function allows the machine code routine to return a value to the **BASIC** program made up from the register contents. For more details of `CALL` and `USR` refer to Chapter 5.

Conditional Assembly and Macros

Working within the **BASIC** environment it is possible to use **BASIC** functions to implement these higher level assembly language structures.

Conditional assembly is a method of varying the code assembled according to a test. All the facilities of **BASIC** are available for setting up the test criteria. Typical applications for conditional assembly include the conditional incorporation of debugging routines and selecting different hardware specific sub-routines from a number of alternatives.

A macro is a group of assembler statements which may be inserted into the assembler program when called. A macro may be thought of as being a type of sub-routine which is used to include a portion of assembler used more than once within a program. A number of statements which are likely to be used more than once can be enclosed within assembler delimiters and placed within either a sub-routine (called using GOSUB and terminated by RETURN), or a function definition or a procedure definition. Using a procedure or a function is the best way to implement macros because the programmer is then able to pass parameters to the macro and the procedure/function name serves to identify the macro.

e.g.:

```
 10 DIM MC% 100
 20 FOR opt%=0 TO 3 STEP 3
 30 P%=MC%
 40 [
 50 OPT opt%
 60 .add CLC                    \ clear carry
 70 LDA &80                     \ A=?&80
 80 ADC &81                     \ AA+?&81+carry
 90 STA &81                     \ ?&81=A
100 OPT FNdebug(TRUE)
110 ]
120 NEXT
130 ?&80=1
30
140 ?&81=2
150 CALL add
160 PRINT'"Result of addition : ";?&81
170 PRINT'"A=&";~?&70,"X&" ~?&71,"Y&";~?&72
180 END
190 DEF FNdebug(switch)
200 IF switch [OPT opt%:STA &70:STX &71:STY &72:]:REM Save registers
210 [OPT opt%:RTS:]
220 =opt%
```

This highly contrived program adds two bytes together. It uses a macro within which conditional assembly occurs. Hanging a function on the end of an **OPT** command enables the programmer to call the macro in a tidy manner. If **FNdebug** is called with the value **TRUE** then some code which saves the registers in zero page is inserted into the program otherwise an **RTS** instruction is inserted. The function returns with the value to which **OPT** was set in the first place. This example indicates how the close inter-relation of the mnemonic assembler with **BASIC** results in a very powerful assembler. The programmer should always remember that **BASIC** is always available as an aid when using the **BASIC** assembler.

User Zero Page

32 bytes of zero page locations are reserved by **BASIC** for the users machine code programs. These locations are from &70 to &8F (inclusive). These are the only zero page locations that a user program (resident in RAM) should use if the program is to be made commercially available or run on a variety of other BBC Microcomputers.

The locations from &0 to &6F which are part of **BASIC**'s zero page workspace are available to the machine code program if **BASIC** is not required while the code is running.

Depending on the nature of the machine code program other zero page locations may be available. See a later chapter for more details.

12. The 6502 Instruction Set

The 6502 registers and abbreviations

Accumulator - A

An 8 bit general purpose register used for all the arithmetic and logical operations.

X Index Register - X

An 8 bit register used as the offset in indexed and pre-indexed indirect addressing modes, or as a counter.

Y Index Register - Y

An 8 bit register used as the offset in indexed and post-indexed indirect addressing modes, or as a counter.

Status Register

An 8 bit register containing various status flags and an interrupt mask. These are:

Carry flag - C

Bit 0, Set if a carry occurs during an add operation and cleared if a borrow occurs during subtraction. Used as a 9th bit in rotate and shift operations.

Zero flag - Z

Bit 1, Set if the result of an operation is zero, otherwise cleared.

Interrupt disable - I

Bit 2, When set, IRQ interrupts are disabled. Set by the processor during interrupts.

Decimal mode flag - D

Bit 3, When set the add and subtract instructions work in binary coded decimal arithmetic. When clear these operations are performed using binary arithmetic.

Break flag - B

Bit 4, This flag is set by the processor during a BRK interrupt. Otherwise this flag is clear.

Unused flag

Bit 5, Unused by the processor.

Overflow flag - V

Bit 6, If, during an operation, there is a carry from bit 6 to bit 7 and no external carry then the overflow flag is set. This flag is also set if there is no carry from bit 6 to bit 7 but there is an external carry.

Negative flag - N

Bit 7, Set if bit 7 of a result is set, otherwise cleared.

Stack Pointer - SP

An 8 bit register which forms the low order byte of the address of the next free stack location (the high order byte of this address is always &1).

Program Counter - PC (PCL, PCH low-byte, high-byte)

A 16 bit register which always contains the address of the next instruction to be executed.

The Assembler Mnemonics

The following section contains a detailed description of each of the operation codes (or instructions) in the 6502 instruction set. The assembler recognises three letter mnemonics which it translates into the 8 bit values which the microprocessor actually takes as its instructions.

Each assembler mnemonic is described on a new page. At the head of the page is the three letter mnemonic which the assembler recognises.

Beneath the heading there is a short phrase indicating the function of the instruction and the derivation of the mnemonic.

A short hand '**BASIC** like' description of the operation is given on the top right of the page. The registers and flags are denoted by the abbreviations given on the previous two pages. The initial 'M' represents the data byte obtained using the selected addressing mode.

A brief description of the instruction and its operation is given beneath the headings.

Any changes to the status register are noted in a list of the status register flags.

All the available addressing modes are listed together with the number of bytes of memory which the instruction and its data will occupy when this mode is used. The number of instruction cycles taken for the execution of the instruction in each addressing mode is also given (1 instruction cycle=0.5 microseconds).

A short example of the use of the instruction within an assembly language routine is given at the bottom of each page.

ADC

Add with Carry A, C = A + M + C

This instruction adds the contents of a memory location to the accumulator together with the carry bit. If overflow occurs the carry bit is set, this enables multiple byte addition to be performed.

Processor Status after use:

C (carry flag): set if overflow in bit 7
Z (zero flag): set if A=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): set if sign bit is incorrect
N (negative flag): set if bit 7 set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)
absolute,Y	3	4 (+1 if page crossed)
(indirect,X)	2	6
(indirect),Y	2	5 (+1 if page crossed)

Example: Add 1 to a 2 byte value in locations &80 and &81

```
CLC         \clear carry flag
LDA #1      \load accumulator with 1
ADC &80     \A=A+?&80, carry set if overflow occurs
STA &80     \place result of addition in &80
LDA #0      \set accumulator to 0 (carry unchanged)
ADC &81     \A=A+?&81+C, add 1 if carry set
STA &81     \store result back in &81
```

AND

Logical AND A = A AND M

A logical AND is performed, bit by bit, on the accumulator contents using the contents of a byte of memory. The truth table for the logical AND is:

Acc. bit	Mem. bit	Result bit
0	0	0
0	1	0
1	0	0
1	1	1

Processor Status after use:

C (carry flag): not affected
Z(zero flag): set if A=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)
absolute,Y	3	4 (+1 if page crossed)
(indirect,X)	2	6
(indirect),Y	2	5 (+1 if page crossed)

Example: Clear the top 4 bits of location &80

```
LDA &80      \load value to be ANDed into A
AND #&F0     \perform AND, (mask=11110000)
STA &80      \load memory with the modified value
```

ASL

Arithmetic Shift Left M = M * 2, C = M7 (or accumulator)

This operation shifts all the bits of the accumulator or memory contents one bit left. Bit 0 is set to 0 and bit 7 is placed in the carry flag. The effect of this operation is to multiply the memory contents by 2 (ignoring 2's complement considerations), setting the carry if the result will not fit in 8 bits.

C ← 76543210 ← 0

Processor Status after use:

C (carry flag): set to old contents of bit 7
Z (zero flag): set if result=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
accumulator	1	2
zero page	2	5
zero page,X	2	6
absolute	3	6
absolute,X	3	7 (+1 if page crossed)

Example: Rapid multiplication of memory contents by 4

```
ASL data     \?data=?data*2
ASL data     \?data=?data*2, gross effect *4.
```

BCC

Branch on Carry Clear Branch if C = 0

This instruction causes a relative jump if the carry flag is clear. The address to which the branch is directed must be within relative addressing range otherwise the assembler will throw up an `Out of range` message.

Used after a CMP instruction this branch occurs when A<DATA.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: Branch if contents of &80 < 100

```
LDA #100     \load accumulator with data
CMP &80      \A-data (comparison)
BCC finish   \goto finish if ?&80<100
```

BCS

Branch on Carry Set Branch if C = 1

A relative branch will occur if the carry flag is set. The branch address given to the assembler must be within relative addressing range.

Used after a CMP instruction this branch occurs when A>=data.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: Branch if contents of X register are greater than or equal to 5

```
CPX #5      \X-5, compare
BCS label   \branch to label if X>=5
```

BEQ

Branch on result zero Branch if Z = 1

This instruction causes a relative branch if the zero flag is set when the instruction is executed. The assembler automatically calculates the relative address from the address given and will cause an error if the address is out of range.

- Used after a CMP instruction this branch occurs if A=data.
- Used after an LDA instruction this branch occurs if A=0.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: Subroutine not used when A=3

```
CMP #3          \A-3, comparison
BEQ over        \if A=3 goto over
JSR anything    \subroutine to be missed if A<>3
.over .....
```

BIT

Test memory bits with accumulator A AND M, N = M7, V = M6

This instruction can be used to test whether one or more specified bits are set. The zero flag is set if the result is 0 otherwise the zero flag is cleared. Bits 7 and 6 of the memory location are transferred to the status register. The `BIT` instruction performs an `AND` operation without storing the result but setting the status flags.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if the result=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): set to bit 6 of memory
N (negative flag): set to bit 7 of memory

Addressing mode	bytes used	cycles
zero page	2	3
absolute	3	4

Example: Test bit 7 of location &8F

```
LDA #&02     \load mask into accumulator (00000010)
BIT flags    \A AND flags, if bit 1=1 then Z=0
BNE flagset \action to be performed if bit 1 set
```

BMI

Branch if negative flag set Branch if N = 1

This relative branch is performed if the result of a previous operation was negative. Relative branch calculations are made by the assembler which will flag an error if an address is given outside the relative addressing range.

Branch occurs after a result which sets bit 7 of the accumulator. (All 8 bit 2's complement negative numbers have this bit set.)

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: Branching if a byte of memory contains a negative number

```
LDA &3010       \load accumulator from memory, N set
                if -Ve
BMI negative    \branch if ?&3010 is negative
```

BNE

Branch on result not zero Branch if Z = 0

This instruction causes a relative branch if the zero flag is clear when the instruction is executed. The assembler automatically calculates the relative address from the address given and will cause an error if the address is out of range.

- Used after a CMP instruction this branch occurs if A<>data.
- Used after an LDA instruction this branch occurs if A<>0.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: Memory location to be written to if it contains zero (i.e. `IF ?&84=0 then ?&84=&7F`)

```
LDA &84      \load memory into A to set flags
BNE round    \if not zero skip the next bit
LDA #&7F     \lead A with value to he written
STA &84      \write to location &84
.round .... \rest of program
```

BPL

Branch on positive result Branch if N = 0

Depending on the state of the negative flag a relative branch will be made. The relative address is calculated by the assembler from an address provided by the programmer. This address must be within the relative addressing range.

Branch occurs after a result which sets accumulator bit 7 to 0.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: A loop which shifts A left until bit 7 is set

```
.loop ASL A        \ shift accumulator 1 bit left
      BPL loop     \ if bit 7 not set then go round again
```

NOTE: This will be an endless loop if A=0 on entry.

BRK

Forced Interrupt PC and P pushed on stack
PCL = ?&FFFE, PCH = ?&FFFF

This instruction forces an interrupt to occur. The processor jumps to the location stored at &FFFE. The program counter is pushed onto the stack followed by the status register. A BRK instruction usually represents an error condition and the BRK handling code is usually an error handling routine. Using machine code in a **BASIC** environment it is possible to use **BASIC**'s error handling facilities. A user BRK handling routine may be implemented, see Vectors section.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): set
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	7

NOTE: A BRK instruction cannot be disabled by setting the interrupt disable flag.

Example: Cause an error if A is greater than 4

```
CMP #5       \A-5. comparison
BCC noerr    \if A<5 then branch round error
BRK          \cause error
.noerr .... \rest of program (or error message)
```

BVC

Branch if overflow clear Branch if V = 0

A relative branch is made if the overflow flag is clear. The relative address calculation is performed by the assembler which will flag an error if given an address out of relative addressing range.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: Branching on overflow when carry is deliberately set

```
ADC &80             \A=A+?&80+C
SEC                 \set carry flag
BVC somewhere       \goto somewhere if no overflow
```

BVS

Branch if overflow set Branch if V = 1

Branch to a relative address if the overflow flag is set. Overflow is generally set when the carry flag is set except when a subtraction is performed. In this case overflow is set when the carry flag is cleared. The address specified in the operand field of the assembler statement must be within the relative addressing range otherwise an assembly error will be flagged.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
relative	2	2 (+1 if branch succeeds +2 if to new page)

Example: Branching if overflow occurs during subtraction

```
SEC         \set the carry flag
LDA #8      \load A with the value 8
SBC &86     \A=A-M (-carry if required)
BVS help    \if overflow has occurred goto help
STA &86     \otherwise put new value in &86
```

NOTE: A BCC instruction would have performed the same purpose in this instance.

CLC

Clear carry flag C = 0

This instruction clears the carry flag. This is often a sensible operation to perform before using an ADC instruction if there is any doubt as to the status of the carry flag.

Processor Status after use:

C (carry flag): cleared
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: Clearing the carry flag before an 8 bit addition

```
CLC                 \clear carry flag
LDA counter         \load first low order byte
ADC increment       \add second low order to it
STA counter         \place new value in counter
```

CLD

Clear decimal flag D = 0

This flag is used to place the 6502 into decimal mode. This instruction returns the processor into non-decimal mode.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): cleared
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: Turn decimal mode off

```
CLD             \No more BCD arithmetic
```

See SED for a detailed description of Binary Coded Decimal Mode.

CLI

Clear interrupt disable flag I = 0

This instruction is used to re-enable interrupts after they have been disabled by setting the interrupt flag. In a machine where the operating system relies heavily on interrupts it is unwise to play around with the interrupt flag without good reason.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): cleared
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: Re-enabled interrupts

```
CLI             \interrupts responded to now
```

CLV

Clear the overflow flag V = 0

This instruction forces the overflow flag to be cleared.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): cleared
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: Explicitly clear the overflow flag

```
CLV         \overflow now clear
```

CMP

Compare memory and accumulator A - M

This is a very useful instruction for comparing the accumulator contents to the contents of a memory location. The status register flags are set according to the result of a subtraction of the memory contents from the accumulator. The accumulator contents are preserved but the status register flags may be used to cause branches depending on the values which were compared.

Processor Status after use:

C (carry flag): set if A greater than or equal to M
Z (zero flag): set if A=M
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)
absolute,Y	3	4 (+1 if page crossed)
(indirect),X	2	6
(indirect,Y)	2	5 (+1 if page crossed)

Examples: Branching on the result of a comparison.

```
The test which if true
is to cause the branch. Code.

A>M   or    M<A         BEQ over (or BEQ P%+4, no label)
                        BCS somewhere

A>=M  or    M<=A        BCS somewhere

A=M   or    M=A         BEQ somewhere

A<=M  or    M>=A        BCC somewhere
                        BEQ somewhere

A<M   or    M>A         BCC somewhere
```

CPX

Compare memory with X register X - M

This instruction performs a subtraction of the contents of the memory location from the contents of the X register, the memory location and the register remain intact but the status register flags are set on the result.

Processor Status after use:

C (carry flag): set if X greater than or equal to M
Z (zero flag): set if X=M
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
absolute	3	4

Example: Clearing an area of memory (max &100 bytes). The number of bytes to be cleared is stored in 'count'.

```
      LDA #0          \set accumulator to 0
      TAX             \set loop index to 0
.loop STA page,X      \write 0 to byte page+X
      INX             \increment loop index
      CPX count       \X-?count, comparison
      BNE loop        \if not equal go round again
```

CPY

Compare memory with Y register Y - M

This instruction performs a subtraction from the Y register of the specified memory location contents. The memory location and the register remain intact but the status register flags are set on the result.

Processor Status after use

C (carry flag): set if Y greater than or equal to M
Z (zero flag): set if Y=M
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
absolute	3	4

Example: Branch if Y=&0D

```
CPY #&0D     \compare Y with &0D/13
BEQ cr       \if Y=13 goto cr
```

DEC

Decrement memory by one M = M - 1

This instruction decrements the value contained in the specified memory location.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if memory contents become 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
zero page	2	5
zero page,X	2	6
absolute	3	6
absolute,X	3	7

Example: Decrement location &2900

```
DEC &2900    \?&2900=?&2900-1
```

DEX

Decrement X register by one M = M - 1

This instruction decrements the contents of the X register by one.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if X becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of X becomes set

Addressing mode	bytes used	cycles
implied	1	2

Example: Decrement X register

```
DEX             \X=X-1
```

DEY

Decrement Y register by one Y = Y - 1

This instruction decrements the contents of the Y register by one.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if Y becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of Y becomes set

Addressing mode	bytes used	cycles
implied	1	2

Example: Decrement Y register

```
DEY             \Y=Y-1
```

EOR

Exclusive OR memory with accumulator A = A EOR M

This instruction performs a bit by bit Exclusive OR of the specified memory location contents with the contents of the accumulator leaving the result in the accumulator. The truth table for the logical EOR operation is:

Acc. bit	Mem. bit	Result bit
0	0	0
0	1	1
1	0	1
1	1	0

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if A becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of A becomes set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)
absolute,Y	3	4 (+1 if page crossed)
(indirect,X)	2	6
(indirect),Y	2	5 (+1 if page crossed)

Example: EOR contents of memory with &FF

```
LDA #&FF \ load accumulator with &FF
EOR temp \ A=A EOR (?temp)
STA temp \ reload memory
```

INC

Increment memory by one M = M + 1

This instruction increments the value contained in the specified memory location.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if memory contents become 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of memory becomes set

Addressing mode	bytes used	cycles
zero page	2	5
zero page,X	2	6
absolute	3	6
absolute,X	3	7

Example: Increment location &80

```
INC &80       \?&80=?&80+1
```

INX

Increment X register by one X = X + 1

This instruction increments the contents of the X register by one.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if X becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of X becomes set

Addressing mode	bytes used	cycles
implied	1	2

Example: Increment X register

```
INX             \X=X+1
```

INY

Increment Y register by one Y=Y+1

This instruction increments the contents of the Y register by one.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if Y becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of Y becomes set

Addressing mode	bytes used	cycles
implied	1	2

Example: Increment Y register

```
INY             \Y=Y+1
```

JMP

Jump to new location PC = new address

This instruction is the machine code equivalent of a `GOTO` statement in **BASIC**. An indirect addressing mode is available where the address for the JMP is contained in memory specified by the address in the operand field (see examples below).

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
absolute	3	3
indirect	3	5

Examples: A direct jump

```
JMP entry          \goto entry
```

An indirect jump (a contrived example)

```
LDA #&00           \A=0
STA &2800          \?&2800=A (address low byte)
LDA #&40           \A=&40
STA &2801          \?&2801=A (address high byte)
JMP (&2800)        \jump to &4000
```

JSR

Jump Subroutine Push current PC onto stack; PC = new address

This instruction is the machine code equivalent of a `GOSUB` statement in **BASIC**. It causes a jump but also saves the current program counter on the stack. The subroutine which is called returns to the part of the program that called it by pulling the saved address and jumping back to it. A subroutine must always be terminated by an RTS instruction which performs the return to the location from which the subroutine was called.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
absolute	3	6

Examples: Using an OS call

```
LDA #ASC"X"
JSR OSWRCH          \print 'X' on screen
```

LDA

Load accumulator from memory A = M

This instruction is used to set the contents of the accumulator to that contained in a specified byte of memory.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if A=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of A set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)
absolute,Y	3	4 (+1 if page crossed)
(indirect,X)	2	6
(indirect),Y	2	5 (+1 if page crossed)

Example: Load accumulator with ASCII value for 'A'

```
LDA #ASC'A'          \A=65
```

LDX

Load X register from memory X = M

This instruction is used to set the contents of the X register to that contained in a specified byte of memory.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if X=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of X set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,Y	2	4
absolute	3	4
absolute,Y	3	4 (+1 if page crossed)

Example: Load X register with contents of location &80

```
LDX &80       \X=?&80
```

LDY

Load Y register from memory Y = M

This instruction is used to set the contents of the Y register to that contained in a specified byte of memory.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if Y=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of Y set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)

Example: Load Y register with contents of location labelled 'data' with an offset in X

```
LDY data,X          \Y=?(data+X)
```

LSR

Logical Shift Right by one bit M = M / 2 (or A)

This instruction causes each bit in the memory location or accumulator to shift one bit right. Bit 7 is set to 0 and the carry flag will be set to the old contents of bit 0. The arithmetic effect of this is to divide the value by 2.

0 → 76543210 → C

Processor Status after use:

C (carry flag):set to bit 0 of operand
Z (zero flag): set if result=0
I (interrupt disable): not affected
D (decimal mode flag):not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): cleared

Addressing mode	bytes used	cycles
accumulator	1	2
zero page	2	5
zero page,X	2	6
absolute	3	6
absolute,X	3	7 (+1 if page crossed)

Example: Shift accumulator contents right one bit

```
LSR A          \C=bit 0, A=A/2
```

NOP

No operation

This is a dummy instruction which has no effect on any memory or register contents except to increment the program counter by one. It can be useful for padding cycles, to ensure timing in loops, or padding code to patch routines that are already assembled.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: A NOP instruction

```
NOP         \this instruction does nothing
```

ORA

OR memory with accumulator A = A OR M

This instruction performs a bit by bit logical OR operation between the contents of the accumulator and the contents of the specified memory and places the result in the accumulator. The truth table for logical OR is:

Acc. bit	Mem. bit	Result bit
0	0	0
0	1	1
1	0	1
1	1	1

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if A=0
I (interrupt disable): not affected
D(decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of A set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)
absolute,Y	3	4 (+1 if page crossed)
(indirect,X)	2	6
(indirect),Y	2	5 (+1 if page crossed)

Example: Set the top 4 bits of the accumulator

```
ORA #&F0          \mask is 1111000, 1 OR anything=1
```

PHA

Push accumulator onto stack Push A

This instruction places the value held in the accumulator onto the stack. This value is accessible using the instruction PLA (pull A from stack).

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	3

Example: Save registers at the beginning of a routine

```
.entry      PHP   \save status register (see below)
            PHA   \save accumulator contents
            TXA   \A=X
            PHA   \save X register contents
            TYA   \A=Y
            PHA   \save Y register contents
            ....  \rest of program
```

PHP

Push Status register onto stack Push P

This instruction places the value held in the status register onto the stack. This value is accessible using the instruction PLP (pull P from stack).

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	3

Example: See the example given for PHA above.

PLA

Pull accumulator off stack Pull A

This instruction loads the accumulator with a value which is pulled from the stack. This is usually a previous accumulator value which has been saved on the stack using a PHA instruction.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if A=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of A set

Addressing mode	bytes used	cycles
implied	1	4

Example: Restore registers at the end of a routine

```
PLA         \pull Y value from stack
TAY         \put it back in Y
PLA         \pull X value from stack
TAX         \put it back in X
PLA         \pull A value from stack
PLP         \restore status register
RTS         \back to calling routine
```

PLP

Pull status register off stack Pull P

This instruction loads the status register with a value which is pulled from the stack. This is usually a previous status register value which has been saved on the stack using a PHP instruction.

Processor Status after use:

C (carry flag): bit 0 from stack
Z (zero flag): bit 1 from stack
I (interrupt disable): bit 2 from stack
D(decimal mode flag): bit 3 from stack
B (break command): bit 4 from stack
V (overflow flag): bit6 from stack
N (negative flag): bit 7 from stack

Addressing mode	bytes used	cycles
implied	1	4

Example: See the example for PLA above.

ROL

Rotate one bit left M = M *2, M0 = C, C = M7 (A or M)

This instruction causes a shift left one bit. The bit shifted out of the byte, bit 7, is placed in the carry flag. The contents of the carry flag are placed in bit 0.

Processor Status after use:

C (carry flag): set to old value of bit 7
Z (zero flag): set if result=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
accumulator	1	2
zero page	2	5
zero page,X	2	6
absolute	3	6
absolute,X	3	7 (+1 if page crossed)

Example: Rotate accumulator contents one bit left

```
ROL A          \A=A rotated left
```

NOTE: The carry flag state should be known before this operation is performed.

ROR

Rotate one bit right M = M / 2, M7 = C, C = M0 (A or M)

This instruction causes a shift right one bit. The bit shifted out of the location, bit 0 is placed in the carry flag. The contents of the carry flag are placed in bit 7.

Processor Status after use:

C (carry flag): set to old value of bit 0
Z (zero flag): set if result=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
accumulator	1	2
zero page	2	5
zero page,X	2	6
absolute	3	6
absolute,X	3	7 (+1 if page crossed)

Example: Reverse the order of bits in a byte

```
.start      STA &80     \store byte in &80
            LDX #8      \set loop count to 8
.loop       ROL &80     \bit 7 of &80 to carry
            ROR A       \carry to bit 8 of A
            DEX         \decrement loop count
            BNE loop    \if not 0 goto loop
            RTS         \exit with A reversed
```

RTI

Return from Interrupt Status register and PC pulled from stack

This instruction is used to return from an interrupt handling routine. When an interrupt occurs the current program counter and status register are pushed onto the stack. These are restored by the RTI instruction.

Processor Status after use:

C (carry flag): bit 0 from stack
Z (zero Hag): bit 1 from stack
I (interrupt disable): bit 2 from stack
D (decimal mode flag): bit 3 from stack
B (break command): bit 4 from stack
V (overflow flag): bit 6 from stack
N (negative flag): bit 7 from stack

Addressing mode	bytes used	cycles
implied	1	6

Example: Instruction at the end of an interrupt handling routine

```
....        \code dealing with the interrupt
RTI         \back to what we were doing before ...
```

RTS

Return from subroutine Pull PC from stack

The RTS instruction is used to terminate the execution of a subroutine. Any routine terminated in this way should be called using a JSR instruction which places a return address on the stack. The top two stack values are placed in the program counter and execution is resumed at the point in the program after the JSR instruction. During a subroutine the same number of items pushed on the stack must be removed before the RTS instruction is reached if the subroutine is to return to the correct address.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag):not affected

Addressing mode	bytes used	cycles
implied	1	6

Example: Last instruction in a subroutine

```
....        \body of subroutine
RTS         \return to calling routine
```

SBC

Subtract memory from accumulator with carry A,C = A – M - (1 - C)

This instruction subtracts the contents of the specified memory from the accumulator contents leaving the result in the accumulator. If the carry flag is used as a 'borrow' source and if clear then an extra unit is subtracted from the accumulator. This enables the 'borrow' to be carried over in multi-byte subtractions (see example below).

Processor Status after use:

C (carry flag): cleared if a borrow occurs
Z (zero flag): set if result=0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): set if the sign of the result is wrong
N (negative flag): set if bit 7 of the result is set

Addressing mode	bytes used	cycles
immediate	2	2
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	4 (+1 if page crossed)
absolute,Y	3	4 (+1 if page crossed)
(indirect,X)	2	6
(indirect),Y	2	5 (+1 if page crossed)

Example: 16 bit value at locations &80 and &81 subtracted from 16 bit value at locations &82 and &83, result at locations &82 and &83.

```
SEC         \ready for any borrow
LDA &80     \low order byte of first value
SBC &82     \A=A-?&82 (borrow may occur)
STA &82     \place result in &82
LDA &81     \high order byte of first value
SBC &83     \A=A-&83(1-C)
STA &83     \place result in &83
```

SEC

Set carry flag C = 1

This instruction is used to set the carry flag. This instruction should be used to set the carry flag prior to a subtraction unless the carry flag has been deliberately left as a 'borrow' from a previous subtraction.

Processor Status after use:

C (carry flag): set
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: Explicit setting of the carry flag

```
SEC             \C=1
```

SED

Set decimal mode D = 1

This instruction is used to place the 6502 in decimal mode. This causes arithmetic operations to be performed in BCD mode.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): set
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Binary Coded Decimal Mode

A binary coded decimal arithmetic mode may be selected by setting the decimal flag in the status register. The binary coded decimal form of representing numbers uses each byte to store a two digit decimal value. Each digit is stored as a binary value in 4 bits (1 nibble). Normally 4 bits can be used to represent numbers in the range 0 to 15. In BCD arithmetic 6 of the values that could be represented in 4 bits are not used. Adding 1 to 9 in BCD will cause the low-nibble to be set to 0 and the high nibble to be set to 1. The carry flag is used to store the carry from the high-nibble.

This is an example of a program which uses BCD arithmetic:

```
10 DIM MC% 100
20 OSWRCH=&FFEE
30 OSRDCH=&FFE0
40 OSNEWL=&FFE7
50 FORopt%=0 TO 3 STEP3
60     P%=MC%
70     [
80     OPT opt%
```

```
 90     .start  SED             \set flag for BCD arithmetic
100             CLC             \clear carry flag
110             LDA     &80     \A=?&80
120             ADC     #1      \A=A+1+C
130             STA     &80     \replace value
140             LDA     &81     \A=?&81
150             ADC     #0      \A=A+0+C
160             STA     &81     \replace value
170             CLD             \clear flag, no more BCD
180             CLC             \clear carry flag
190             LDX     #2      \set loop index
200     .loop   DEX             \decrement index
210             LDA     #&FO    \mask for high-nibble
220             AND     &80,X   \A=A AND X?&80
230             LSR     A:LSR A:LSR A:LSR A
240                             \move high-nibble to low nibble
250             ADC     #&30    \add value to ASCO"
260             JSR     OSWRCH  \print value
270             LDA     #&F     \mask for low-nibble
280             AND     &80,X   \A=A AND X?&80
290             ADC     #&30    \add value to ASC"0"
300             JSR     OSWRCH  \print number
310             CPX     #0      \has index reached 0
320             BNE     loop    \if not, go round again
330             LDA     #&D     \A=carriage return value
340             JSR     OSWRCH  \perform carriage return (no LF)
350             JSR     OSRDCH  \A=GET
360             CMP     #&OD    \was it RETURN
370             BNE     start   \if not, back to the start
380             JSR     OSNEWL  \carriage return and line feed
390             RTS             \back to BASIC
400             ]
410             NEXT
420     !&80=0
430     PRINT''"Binary Coded Decimal"''
440     PRINT"press key to add 1"
450     PRINT"press RETURN to exit"''
460     CALL start
```

This program could be altered to subtract 1 each time a key is pressed by changing line 100 to SEC and changing the ADC instructions in lines 120 and 150 to SBC instructions.

The decimal flag must always be cleared before using operating system routines.

There is no standard representation of negative numbers using BCD. In order to implement more complex arithmetic including floating point applications the programmer must define his own conventions and number formats.

SEI

Set interrupt disable flag I = 1

This instruction is used to set the interrupt disable flag. When this flag is set maskable interrupts cannot occur.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): set
D (decimal mode flag):not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: Disable interrupts

```
SEI          \No maskable interrupts
```

STA

Store accumulator contents in memory M = A

This instruction is used to copy the contents of the accumulator into a memory location specified in the operand field.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
zero page	2	3
zero page,X	2	4
absolute	3	4
absolute,X	3	5
absolute,Y	3	5
(indirect,X)	2	6
(indirect),Y	2	6

Example: Store accumulator in location 'save' + Y offset

```
STA save,y          \?(save+Y)=A
```

STX

Store X contents in memory M = X

This instruction is used to copy the contents of the X register into a memory location.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
zero page	2	3
zero page,Y	2	4
absolute	3	4

Example: Store X in location &80

```
STX &80          \?&80=X
```

STY

Store Y contents in memory M=Y

This instruction is used to copy the contents of the Y register into a memory location.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
zero page	2	3
zero page,X	2	4
absolute	3	4

Example: Store Y in location &5FF0

```
STY &5FF0          \?&5FF0=Y
```

TAX

Transfer A to X X = A

This instruction is used to copy the contents of the accumulator to the X register.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if X becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of X is set

Addressing mode	bytes used	cycles
implied	1	2

Example: Transfer contents of A to X

```
TAX             \X=A
```

TAY

Transfer A to Y Y = A

This instruction is used to copy the contents of the accumulator to the Y register.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if Y becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of Y is set

Addressing mode	bytes used	cycles
implied	1	2

Example: Transfer contents of A to Y

```
TAY             \Y=A
```

TSX

Transfer S to X X = S

This instruction is used to copy the contents of the stack pointer to the X register.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if X becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of X is set

Addressing mode	bytes used	cycles
implied	1	2

Example: Transfer contents of S to X

```
TSX             \X=S
```

TXA

Transfer X to A A=X

This instruction is used to copy the contents of the X register to the accumulator.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if A becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of A is set

Addressing mode	bytes used	cycles
implied	1	2

Example: Transfer contents of X to A

```
TXA             \A=X
```

TXS

Transfer X to S S = X

This instruction is used to copy the contents of the X register to the stack pointer.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): not affected
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): not affected

Addressing mode	bytes used	cycles
implied	1	2

Example: Transfer contents of X to S

```
TXS             \S=X
```

TYA

Transfer Y to A A = Y

This instruction is used to copy the contents of the Y register to the accumulator.

Processor Status after use:

C (carry flag): not affected
Z (zero flag): set if A becomes 0
I (interrupt disable): not affected
D (decimal mode flag): not affected
B (break command): not affected
V (overflow flag): not affected
N (negative flag): set if bit 7 of A is set

Addressing mode	bytes used	cycles
implied	1	2

Example: Transfer contents of Y to A

```
TYA             \A=Y
```

13. Operating System calls

Input/Output

The input device from which characters may be fetched can be selected using the OSBYTE call with A=2 (*FX 2). Input may be selected from the keyboard and/or RS423.

Output may be channelled to any combination of the following destinations: Screen, Printer, RS423 or Spooled file. Selection of destination is achieved using OSBYTE call with A=3 (*FX 3).

See the OSBYTE call section (chapter 14) for a full description of these OSBYTE calls.

OSWRCH - Write character to currently selected O/P stream.

Call address &FFEE

Indirected through &20E

This routine writes the character given in the accumulator to the currently selected output stream or streams.

NOTE: Unrecognised VDU commands are passed to a vector at location &226.

After an OSWRCH call:

- A, X and Y are preserved.
- C, N, V and Z are undefined.
- The interrupt status is preserved (though it may be enabled during a call).

Non-Vectored OSWRCH

Call address &FFCB

This routine is normally used by OSWRCH and the call address is contained in the OSWRCH vector on reset. The non-vectored OSWRCH routine is believed to be used

by the Tube system. This routine has not been documented by Acorn and should be used with caution.

OSRDCH - Read character from currently selected I/P stream.

Call address &FFE0

Indirected through &210

This routine reads a character from the currently selected input stream and returns the character read in the accumulator.

After an OSRDCH call:

- C=0 indicates that a valid character has been read.
- C=1 flags an error condition, A contains an error number.
- If an escape condition occurs then A=&1B (27) and C=1, if detected an escape condition must be acknowledged using an OSBYTE call with A=&7E (126).
- X and Y are preserved.
- N, V and Z are undefined.
- The interrupt status is preserved (though interrupts may be enabled during a call).

Non vectored OSRDCH

Call address &FFC8

This routine is normally used by OSRDCH and the call address is placed in the OSRDCH vector on reset. The non-vectored OSRDCH is believed to be used by the Tube system. This call has not been documented by Acorn and should be used with caution.

OSNEWL - Write a newline to selected output stream.

Call address &FFE7

Not indirected

This routine uses OSWRCH to write a linefeed (ASCII &0A/ 10) followed by a carriage-return (ASCII &0D/13) to the currently selected output stream(s).

After an OSNEWL call:

- A=&0D (13)
- X and Y are preserved.
- C, N, V and Z are undefined.
- Interrupt status is preserved (though it may be enabled during a call).

OSASCI - Write character routine where OSNEWL is called when A=&0D (13).

Call address &FFE3

Not indirected

This routine performs an OSWRCH call with the accumulator contents unless called with accumulator contents of &0D (13) when an OSNEWL call is performed.

After an OSASCI call:

- A, X and Y are preserved.
- C, N, V and Z are undefined.
- Interrupt status is preserved (though interrupts may be enabled during a call).

Main VDU character output entry point

Call address &FFBC

This is the entry point for raw VDU character processing. On entry the accumulator contains the character to be written to the VDU drivers. Any settings of OSBYTE &3/`*FX 3` are totally ignored and no output will go to any other destination

(except characters preceded by `VDU 1` which will be sent to the printer if the printer has previously been enabled with a `VDU 2`). This call has not been documented by Acorn. There will normally be no need to use this routine as OSWRCH with the appropriate `*FX3` call can be used to the same effect.

GSINIT - General string input initialise routine.

Call address &FFC2

The GSINIT and GSREAD routines are used by the operating system to process strings used for commands such as `*KEY` and `*LOAD`. The advantage of using this system for reading strings is that an escape sequence can be used to introduce control characters which would otherwise be difficult to type in directly from the keyboard.

This routine should be used to initialise a string which is to be used for input using the GSREAD routine (see below). The routine requires locations &F2 and &F3 plus a Y register offset to specify the string address. The string need not be enclosed by quotation marks. GSINIT must be used not only to strip off leading spaces but also to set up an information byte in zero page which indicates the termination character and if the string is surrounded by quotation marks.

If the carry flag is clear on entry then the first space or carriage return or second quotation mark will be considered as the terminating character for the string. If the carry flag is set then only a carriage return or a second quotation mark will be considered as the terminal character.

On exit:

- Y contains the offset of the first non-blank character from the address contained in &F2 and &F3.
- A contains the first non-blank character (as returned by the first call of GSINIT.
- Z flag is set if the string is a null string

This routine has not been documented by Acorn but has been used in applications software.

GSREAD Read character from string input.

Call address &FFC5

This routine should only be used following a GSINIT call. GSREAD should be entered with Y set to either the Y value following a GSINIT call or following a previous GSREAD call. Locations &F2 and &F3 should contain the address of the start of the string (i.e. should not have been altered since the last GSINIT call).

On exit:

- A contains the character read from the string.
- Y contains the index for the next character to be read.
- Carry flag is set if the end of string is reached.
- X is preserved.

This routine has not been documented by Acorn but has been used in applications software.

OSRDRM - Read byte in paged ROM.

Call address &FFB9

On entry:

- Y=ROM number.
- Locations &F6 and &F7 should contain the address of the byte to be read.

On exit:

- A contains the value of the byte read.

This routine has not been documented by Acorn but has been used in applications software.

OSEVEN - Generate an event.

Call address &FFBF

This call generates or causes an event. The event number should be placed in the Y register when this routine is called. The accumulator contents are transferred to the Y register and the event number is placed in the accumulator when the event handling routine is entered.

This routine has not been documented by Acorn and should be used with caution.

OSCLI - Passes line of text to the Command Line Interpreter.

Call address &FFF7

Indirected through &208

This routine passes a line of text to the command line interpreter which decodes and executes any command recognised.

On entry:

- X and Y should point to a line of text (X= low-byte, Y= high-byte).
- The line of text should be terminated by a carriage return character (ASCII &0D/13)

After an OSCLI call:

- A, X, Y, C, N, V and Z are undefined.
- Interrupt status is preserved but interrupts may be enabled during a call.

14. *FX and OSBYTE calls

The OSBYTE call to the operating system is a powerful and flexible way of invoking many of the available operating system facilities. The `*FX` command can be used to make OSBYTE calls from **BASIC** programs or directly from the keyboard.

OSBYTE - OS call specified by the contents of A taking parameters in X and Y

Call address &FFF4

Indirected through &20A

On entry:

- A selects an OSBYTE routine
- X contains an OSBYTE parameter
- Y contains an OSBYTE parameter

Any OSBYTE calls which are not recognised by the operating system will be offered to paged ROMs. If the unrecognised OSBYTE is not claimed by a paged ROM then a `Bad command` error will be issued (error number 254).

All the OSBYTE calls recognised by the operating system are described in detail in the following pages. The description for each call includes details of the entry parameters required and the state of the registers on exit. All OSBYTE calls may be made using the `*FX` command, but it is not always appropriate to do so (i.e. those calls returning values in the X and Y registers). Where it is appropriate to use a `*FX` command this has been indicated. Preceding the full OSBYTE descriptions is a complete summary of the OSBYTE calls in a list.

OSBYTE calls &A6/166 to &FF/255 can be used to read or write operating system status flags or variables. In OS 1.20 these memory locations extend from &236 to &28F. The action of these calls is to replace the contents of the specified location with:

(<old value> AND Y) EOR X

- To read a location set X=0, Y=&FF.
- To write a location set X=value, Y=0.

On exit:

- X=old value, Y=value of next location

Many of these calls repeat the function of lower value OSBYTEs.

NOTE: These equivalent calls are not guaranteed to have an identical effect when used to set flags or OS variables, and other calls are of no practical use, these are included for completeness.

OSBYTE/*FX Call Summary

dec.	hex.	function
0	0	Print operating system version
1	1	User OSBYTE call, read/write location &281
2	2	Select input stream
3	3	Select output stream
4	4	Enable/disable cursor editing
5	5	Select printer destination
6	6	Set character ignored by printer
7	7	Set RS423 baud rate for receiving data
8	8	Set RS423 baud rate for data transmission
9	9	Set flashing colour mark state duration
10	A	Set flashing colour space state duration
11	B	Set keyboard auto-repeat delay interval
12	C	Set keyboard auto-repeat rate
13	D	Disable events
14	E	Enable events
15	F	Flush selected buffer class
16	10	Select ADC channels to be sampled
17	11	Force an ADC conversion
18	12	Reset soft keys
19	13	Wait for vertical sync
20	14	Explode soft character RAM allocation
21	15	Flush specific buffer

OSBYTE/*FX calls 22 (&15) to 116 (&74) are not used by OS1.20

114	72	select shadow screen mode (OS 2.00+ only)
117	75	Read VDU status
118	76	Reflect keyboard status in LEDs
119	77	Close any SPOOL or EXEC files
120	78	Write current keys pressed information
121	79	Perform keyboard scan
122	7A	Perform keyboard scan from 16 (&10)
123	7B	Inform OS, printer driver going dormant
124	7C	Clear ESCAPE condition
125	7D	Set ESCAPE condition
126	7E	Acknowledge detection of ESCAPE condition

127	7F	Check for EOF on an open file
128	80	Read ADC channel or get buffer status
129	81	Read key with time limit
130	82	Read machine high order address
131	83	Read top of OS RAM address (OSHWM)
132	84	Read bottom of display RAM address (HIMEM)
133	85	Read bottom of display address for a given MODE
134	86	Read text cursor position (POS and VPOS)
135	87	Read character at cursor position
136	88	Perform *CODE
137	89	Perform *MOTOR
138	8A	Insert value into buffer
139	8B	Perform *OPT
140	8C	Perform *TAPE
141	8D	Perform *ROM
142	8E	Enter language ROM
143	8F	Issue paged ROM service request
144	90	Perform *TV
145	91	Get character from buffer
146	92	Read from FRED, 1 MHz bus
147	93	Write to FRED, 1 MHz bus
148	94	Read from JIM, 1 MHz bus
149	95	Write to JIM, 1 MHz bus
150	96	Read from SHEILA, mapped I/O
151	97	Write to SHEILA, mapped I/O
152	98	Examine buffer status
153	99	Insert character into input buffer
154	9A	Write to video ULA control register and copy
155	9B	Write to video ULA palette register and copy
156	9C	Read/write 6850 control register and copy
157	9D	Fast Tube BPUT
158	9E	Read from speech processor
159	9F	Write to speech processor
160	A0	Read VDU variable value

OSBYTE/*FX calls 161(&A1) to 165(&A5) are not used by OS1.20

166	A6	Read start address of OS variables (low byte)
167	A7	Read start address of OS variables (high byte)
168	A8	Read address of ROM pointer table (low byte)
169	A9	Read address of ROM pointer table (high byte)
170	AA	Read address of ROM information table (low byte)
171	AB	Read address of ROM information table (high byte)
172	AC	Read address of key translation table (low byte)
173	AD	Read address of key translation table (high byte)
174	AE	Read start address of OS VDU variables (low byte)
175	AF	Read start address of OS VDU variables (high byte)
176	B0	Read/write CFS timeout counter
177	B1	Read/write input source
178	B2	Read/write keyboard semaphore
179	B3	Read/write primary OSHWM
180	B4	Read/write current OSHWM
181	B5	Read/write RS423 mode
182	B6	Read character definition explosion state
183	B7	Read/write cassette/ROM filing system switch
184	B8	Read RAM copy of video ULA control register
185	B9	Read RAM copy of video ULA palette register
186	BA	Read/write ROM number active at last BRK (error)
187	BB	Read/write number of ROM socket containing **BASIC**
188	BC	Read current ADC channel
189	BD	Read/write maximum ADC channel number
190	BE	Read ADC conversion type
191	BF	Read/write RS423 use flag
192	C0	Read RS423 control flag
193	C1	Read/write flash counter
194	C2	Read/write mark period count
195	C3	Read/write space period count
196	C4	Read/write keyboard auto-repeat delay
197	C5	Read/write keyboard auto-repeat period
198	C6	Read/write *EXEC file handle
199	C7	Read/write *SPOOL file handle
200	C8	Read/write ESCAPE, BREAK effect
201	C9	Read/write Econet keyboard disable
202	CA	Read/write keyboard status byte
203	CB	Read/write RS423 handshake extent

204	CC	Read/write RS423 input suppression flag
205	CD	Read/write cassette/RS423 selection flag
206	CE	Read/write Econet OS call interception status
207	CF	Read/write Econet OSRDCH interception status
208	D0	Read/write Econet OSWRCH interception status
209	D1	Read/write speech suppression status
210	D2	Read/write sound suppression status
211	D3	Read/write BELL channel
212	D4	Read/write BELL envelope number/amplitude
213	D5	Read/write BELL frequency
214	D6	Read/write BELL duration
215	D7	Read/write startup message and !BOOT options
216	D8	Read/write length of soft key string
217	D9	Read/write number of lines printed since last page
218	DA	Read/write number of items in VDU queue
219	DB	Read/write TAB character value
220	DC	Read/write ESCAPE character value
221	DD	Read/write character &CO to &CF status
222	DE	Read/write character &DO to &DF status
223	DF	Read/write character &EO to &EF status
224	E0	Read/write character &FO to &FF status
225	E1	Read/write function key status
226	E2	Read/write SHIFT+ function key status
227	E3	Read/write CTRL+function key status
228	E4	Read/write CTRL+SHIFT+function key status
229	E5	Read/write ESCAPE key status
230	E6	Read/write flags determining ESCAPE effects
231	E7	Read/write JRQ bit mask for user 6522
232	E8	Read/write IRQ bit mask for 6850
233	E9	Read/write IRQ bit mask for system 6S22
234	EA	Read flag indicating Tube presence
235	EB	Read flag indicating speech processor presence
236	EC	Read/write write character destination status
237	ED	Read/write cursor editing status
238	EE	Read/write location &27E, not used by 0S 1.20
239	EF	Read/write location &27F, not used by 0S 1.20
240	F0	Read/write location &280, not used by 0S 1.20
241	F1	Read/write location &281, used by *FX 1

242	F2	F2 Read RAM copy of serial processor ULA
243	F3	Read/write timer switch state
244	F4	Read/write soft key consistency flag
245	F5	Read/write printer destination flag
246	F6	Read/write character ignored by printer
247	F7	Read/write first byte of BREAK intercept code
248	F8	Read/write second byte of BREAK intercept code
249	F9	Read/write third byte of BREAK intercept code
250	FA	Read/write location &28A, not used by OS1.20
251	FB	Read/write location &28B, not used by OS1.20
252	FC	Read/write current language ROM number
253	FD	Read/write last BREAK type
254	FE	Read/write available RAM
255	FF	Read/write start up options

Detailed description of OSBYTE calls

OSBYTE &00 (0) *FX 0

Identify Operating System version

Entry parameters:

- X=0 Execute BRK with a message giving the O.S. type
- X<>0 RTS with O.S. type returned in X

On exit:

- X=0, OS 1.00
- X=1, OS 1.20
- A and Y are preserved
- C is undefined

OSBYTE &01 (1) *FX 1

Read/write the user flag

Entry parameters: The user flag is replaced by:

- (<old value> AND Y) EOR X

i.e. Y=0 for write, Y=&FF for read

On exit:

- X=old value

This call uses OSBYTE call with A=&F1 (241). This OSBYTE call is left free for user applications and is not used by the operating system. The user flag is stored in location &281 and its default value is 0.

OSBYTE &02 (2) *FX 2

Select input stream

Entry parameters: X determines input device(s)

`*FX 2,0`	X=0	keyboard selected, RS423 disabled
`*FX 2,1`	X=1	RS423 selected and enabled
`*FX 2,2`	X=2	keyboard selected, RS423 enabled

Default: `*FX 2,0`

On exit:

- X=0 if previous input was from the keyboard
- X=1 if previous input was from RS423

After call:

- A is preserved
- Y and C are undefined

OSBYTE&03 (3) *FX3

Select output stream

Entry parameters: X determines output device(s), Y=0

- BIT 0 - Enables RS423 driver
- BIT 1 - Disables VDU driver
- BIT 2 - Disables printer driver
- BIT 3 - Enables printer, independent of `<CTRL>` `<B>` or `<C>`
- BIT 4 - Disables spooled output
- BIT 5 - Not used
- BIT 6 -Disables printer driver unless the character is preceded by a `VDU 1` (or equivalent)
- BIT 7- Not used

`*FX 3,0` selects the default output options which are:

- RS423 disabled
- VDU enabled
- Printer enabled (if selected by `VDU 2`)
- Spooled output enabled (if selected by `*SPOOL`)

This OSBYTE call uses OSBYTE call with A=&EC (236). It is thus possible to set Y as a bit mask and only change those bits which are required.

After call:

- A is preserved
- X contains the old `*FX 3` status
- Y and C are undefined

OSBYTE &04 (4) *FX 4

Enable/disable cursor editing

Entry parameters: X determines editing keys' status, Y=0

`*FX 4,0` X=0 Enable cursor editing (default setting)

`*FX 4,1` X=1 Disable cursor editing

The cursor control keys will return the following codes:

- COPY &87 (135)
- LEFT &88 (136)
- RIGHT &89 (137)
- DOWN &8A (138)
- UP &8B (139)

`*FX 4,2` X=2 Disable cursor editing and make the keys act as soft keys with the following soft key association numbers:

- COPY 11
- LEFT 12
- RIGHT 13

- DOWN 14
- UP 15

After call:

- A is preserved
- X contains the previous `*FX 4` setting
- Y and C are undefined

OSBYTE &05 (5) *FX 5

Select printer destination

Entry parameters: X determines print destination

`*FX 5,0`	X=0	Printer sink (printer output ignored)
`*FX 5,1`	X=1	Parallel output (default setting)
`*FX 5,2`	X=2	RS423 output (will act as sink if RS423 is enabled using OSBYTE with A=3)
`*FX 5,3`	X=3	User printer routine
`*FX 5,4`	X=4	Net printer
`*FX 5,5-255`	X=5-255	User printer routine

After call:

- A is preserved
- X contains the previous `*FX 5` setting
- Y and C are undefined
- Interrupts are enabled by this call
- This call is not reset to default by a soft break

OSBYTE &06 (6) *FX 6

Set character ignored by printer

Entry parameters: X contains the character value to be ignored

`*FX 6,10`	X=10	This prevents LINE FEED characters being sent to the printer, unless preceded by `VDU 1` (this is the default setting)

After call:

- A is preserved
- X contains the previous `*FX 6` setting
- Y and C are undefined

OSBYTE &07 (7) *FX 7

Set RS423 baud rate for receiving data

Entry parameters: X determines transmission rate

`*FX 7,1`	X=1	75 baud transmit
`*FX 7,2`	X=2	150 baud transmit
`*FX 7,3`	X=3	300 baud transmit
`*FX 7,4`	X=4	1200 baud transmit
`*FX 7,5`	X=5	2400 baud transmit
`*FX 7,6`	X=6	4800 baud transmit
`*FX 7,7`	X=7	9600 baud transmit
`*FX 7,8`	X=8	19200 baud transmit

After call:

- A is preserved
- X and Y contain the old serial ULA register contents
- C is undefined

OSBYTE &08 (8) *FX 8

Set RS423 baud rate for data transmission

Entry parameters: X determines transmission rate

`*FX 8,1`	X=1	75 baud transmit
`*FX 8,2`	X=2	150 baud transmit
`*FX 8,3`	X=3	300 baud transmit
`*FX 8,4`	X=4	1200 baud transmit
`*FX 8,5`	X=5	2400 baud transmit
`*FX 8,6`	X=6	4800 baud transmit
`*FX 8,7`	X=7	9600 baud transmit
`*FX 8,8`	X=8	19200 baud transmit

After call:

- A is preserved
- X and Y contain the old serial ULA register contents
- C is undefined

OSBYTE &09 (9) *FX 9

Set duration of the mark state of flashing colours (Duration of first named colour)

Entry parameters: X determines length of duration, Y=0

`*FX 9,0`	X=0	Sets mark duration to infinity. Forces mark state if space is set to 0.
`*FX 9,n`	X=n	Sets mark duration to n centiseconds. (n=25 is the default setting)

After call:

- A and X are preserved
- Y contains the old mark duration
- C is undefined

OSBYTE &0A (10) *FX 10

Set duration of the space state of flashing colours (Duration of second named colour)

Entry parameters: X determines length of duration, Y=0

`*FX 10,0`	X=0	Sets space duration to infinity. Forces mark state if space is set to 0.
`*FX 10,n`	X=n	Sets space duration to n centiseconds. (n=25 is the default setting)

After call:

- A and X are preserved
- Y contains the old space duration
- C is undefined

OSBYTE &0B (11) *FX 11

Set keyboard auto-repeat delay

Entry parameters: X determines delay before repeating starts, Y=0

`*FX 11,0`	X=0	Disables auto-repeat facility
`*FX 11,n`	X=n	Sets delay to n centiseconds (n=32 is the default setting)

After call:

- A is preserved
- X contains the old setting
- Y and C are undefined

OSBYTE &0C (12) *FX 12

Set keyboard auto-repeat rate

Entry parameters: X determines auto - repeat periodic interval, Y=0

`*FX 12,0`	X=0	Resets delay and repeat to default values
`*FX 12,n`	X=n	Sets repeat interval to n centiseconds (n=8 is the default setting)

After call:

- A is preserved
- X contains the old `*FX 12` setting
- Y and C are undefined

OSBYTE &0D (13) *FX 13

Disable events

Entry parameters: X contains the event code, Y=0

`*FX 13,0`	X=0	Disable output buffer empty event
`*FX 13,1`	X=1	Disable input buffer full event
`*FX 13,2`	X=2	Disable character entering buffer event
`*FX 13,3`	X=3	Disable ADC conversion complete event
`*FX 13,4`	X=4	Disable start of vertical sync event
`*FX 13,5`	X=5	Disable interval timer crossing 0 event
`*FX 13,6`	X=6	Disable ESCAPE pressed event
`*FX 13,7`	X=7	Disable RS423 error event
`*FX 13,8`	X=8	Disable network error event
`*FX 13,9`	X=9	Disable user event

After call:

- A is preserved
- X and Y contain the old enable state (0= disabled)
- C is undefined

OSBYTE &0E (14) *FX 14

Enable events

Entry parameters: X contains the event code, Y not important

`*FX 14,0`	X=0	Enable output buffer empty event
`*FX 14,1`	X=1	Enable input buffer full event
`*FX 14,2`	X=2	Enable character entering buffer event
`*FX 14,3`	X=3	Enable ADC conversion complete event
`*FX 14,4`	X=4	Enable start of vertical sync event
`*FX 14,5`	X=5	Enable interval timer crossing 0 event
`*FX 14,6`	X=6	Enable ESCAPE pressed event
`*FX 14,7`	X=7	Enable RS423 error event
`*FX 14,8`	X=8	Enable network error event
`*FX 14,9`	X=9	Enable user event

After call:

- A is preserved
- X and Y contain the old enable state (>0=enabled)
- C is undefined

OSBYTE &0F (15) *FX 15

Flush selected buffer class

Entry parameters: X value selects class of buffer

- X=0 All buffers flushed
- X<>0 Input buffer flushed only

See OSBYTE call &16/`*FX 21`

After call:

- Buffer contents are discarded
- A is preserved
- X, Y and C are undefined

OSBYTE &10 (16) *FX 16

Select ADC channels which are to be sampled

Entry parameters: X value selects number of channels sampled

`*FX 16,0`	X=0	Sampling disabled
`*FX 16,n`	X=n	Number of channels to be sampled (n must be in the range 0 to 4, if greater then set to 4)

After call:

- A is preserved
- X contains the old `*FX 16` value

OSBYTE &11 (17) *FX 17

Force an ADC conversion

Entry parameters: X value specifies ADC channel

`*FX 17,n`	X=n	Force ADC conversion on channel n (if n>4 then n=4)

See OSBYTE with A=&80 (128) also

After call:

- A is preserved
- X is preserved if it is in the range 0 to 4 otherwise it is returned containing the value 4
- Y and C are undefined

OSBYTE &12 (18) *FX 18

Reset soft keys

No parameters

This call clears the soft key buffer so the character strings are no longer available

After call:

- A and Y are preserved
- X and C are undefined

OSBYTE &13 (19) *FX 19

Wait for vertical sync

No parameters

This call forces the machine to wait until the start of the next frame of the display. This occurs 50 times per second on the UK BBC Microcomputer and can be used for timing or animation.

NOTE: User trapping of IRQ1 may stop this call from working.

After call:

- A is preserved
- X, Y and C are undefined

OSBYTE &14 (20) *FX 20

Explode soft character RAM allocation

Entry parameters: X value explodes/implodes memory allocation

In the default state 32 characters may be user defined using the `VDU 23` statement from **BASIC** (or the OSWRCH call in machine code). These characters use memory from &C00 to &CFF. Printing ASCII codes in the range 128 (&80) to 159 (&9F) will cause these user defined characters to be printed up (these characters will also be printed out for characters in the range &A0-&BF, &C0-&DF, &E0-&FF). In this state the character definitions are said to be IMPLODED.

If the character definitions are EXPLODED then ASCII characters 128 (&80) to 159 (&9F) can be defined as before using `VDU 23` and memory at &C00. Exploding the character set definitions enables the user to uniquely define characters 32 (&20) to 255 (&FF) in steps of 32 extra characters at a time. The operating system

must allocate memory for this which it does using memory starting at the 'operating system high-water mark' (OSHWM). This is the value to which the **BASIC** variable `PAGE` is usually set and so if a totally exploded character set is to be used in **BASIC** then `PAGE` must be reset to OSHWM+&600 (i.e. `PAGE=PAGE+&600`).

ASCII characters 32 (&20) to 128 (&7F) are defined by memory within the operating system ROM when the character definitions are imploded.

See OSBYTE &83 (131) for details about reading OSHWM from machine code.

The memory allocation for ASCII codes in the expanded state is as follows:

		ASCII code	Memory allocation
`*FX 20,0`	X=0	&80-&8F	&C00-&CFF (imploded)
`*FX 20,1`	X=1	&A0-&BF	OSHWM-OSHWM+&FF (+above)
`*FX 20,2`	X=2	&C0-&DF	OSHWM+&100- OSHWM+&1FF (+above)
`*FX 20,3`	X=3	&E0-&FF	OSHWM+&200- OSHWM+&2FF (+above)
`*FX 20,4`	X=4	&20-&3F	OSHWM+&300- OSHWM+&3FF (+above)
`*FX 20,5`	X=5	&40-&5F	OSHWM+&400-OSHWM+&4FF (+above)
`*FX 20,6`	X=6	&60-&7F	OSHWM+&500-OSHWM+&5FF (+above)

See also OSBYTE call with A=&B6 (182).

After call:

- A is preserved
- X contains the new OSHWM (high byte)
- Y and C are undefined

OSBYTE &15 (21) *FX 21

Flush specific buffer

Entry parameters: X determines the buffer to be cleared

`*FX 21,0`	X=0	Keyboard buffer emptied
`*FX 21,1`	X=1	S423 input buffer emptied
`*FX 21,2`	X=2	S423 output buffer emptied
`*FX 21,3`	X=3	Printer buffer emptied
`*FX 21,4`	X=4	Sound channel 0 buffer emptied
`*FX 21,5`	X=5	Sound channel 1 buffer emptied
`*FX 21,6`	X=6	Sound channel 2 buffer emptied
`*FX 21,7`	X=7	Sound channel 3 buffer emptied
`*FX 21,8`	X=8	Speech buffer emptied

See also OSBYTE calls with A=&0F (*FX15) and A=&80 (128)

After call:

- A and X are preserved
- Y and C are undefined

OSBYTE &72 (114) *FX 114

Select Shadow screen mode

Entry: X determines shadow screen mode

- 0 = Shadow mode off
- 1 = Shadow mode on

This call is only applicable to OS versions 2.00 and later – i.e. the Model B+

OSBYTE &75 (117)

Read VDU status

No entry parameters

On exit the X register contains the VDU status. Information is conveyed in the following bits:

Bit 0 Printer output enabled by a `VDU 2`
Bit 1 Scrolling disabled
Bit 2 Paged scrolling selected
Bit 3 Software scrolling selected i.e. text window
Bit 4 not used
Bit 5 Printing at the graphics cursor enabled by `VDU 5`
Bit 6 Set when input and output cursors are separated (i.e. cursor editing mode).
Bit 7 Set if VDU is disabled by a `VDU 21`

After call:

- A and Y are preserved
- C is undefined

OSBYTE &76 (118)

Reflect keyboard status in keyboard LEDs

This call reflects the keyboard status in the state of the keyboard LEDs, and is normally used after the status has been changed by OSBYTE &CA/202.

On exit:

- A is preserved
- X has bit 7 is set if CTRL is pressed
- Y is undefined

OSBYTE &77 (119) *FX 119

Close any SPOOL or EXEC files

This call closes any open files being used as ***SPOOL**ed output or ***EXEC**d input to be closed. This call also performs a paged ROM call with A=&10 (16).

On exit:

- A is preserved
- X, Y and C are undefined

OSBYTE &78 (120) *FX 120

Write current keys pressed information

The operating system operates a two key roll-over for keyboard input (recognising a second key press even when the first key is still pressed). There are two zero page locations which contain the values of the two key-presses which may be recognised at any one time. If no keys are pressed, location &EC contains 0 and location &ED contains 0. If one key is pressed, location &EC contains the internal key number + 128 (see table below for internal key numbers) and location &ED contains 0. If a second key is pressed while the original key held down, location &EC contains the internal key number + 128 of the most recent key pressed and location &ED contains the internal key number + 128 of the first key pressed.

Internal Key Numbers

hex.	dec.	key	hex.	dec.	key
&00	0	SHIFT	&40	64	CAPS LOCK
&01	1	CTRL	&41	65	A
&02	2	bit 7	&42	66	X
&03	3	bit 6	&43	67	F
&04	4	bit 5	&44	68	Y
&05	5	bit 4	&45	69	J
&06	6	bit 3	&46	70	K
&07	7	bit 2	&47	71	@
&08	8	bit 1	&48	72	:
&09	9	bit 0	&49	73	RETURN
&10	16	Q	&50	80	SHIFT LOCK
&11	17	3	&51	81	S
&12	18	4	&52	82	C
&13	19	5	&53	83	G
&14	20	f4	&54	84	H
&15	21	8	&55	85	N
&16	22	f7	&56	86	L
&17	23	-	&57	87	;
&18	24	A	&58	88	]
&19	25	LEFT CURSOR	&59	89	DELETE
&20	32	f0	&60	96	TAB
&21	33	W	&61	97	Z
&22	34	E	&62	98	SPACE
&23	35	I	&63	99	V
&24	36	7	&64	100	B
&25	37	9	&65	101	M
&26	38	1	&66	102	,
&27	39	0	&67	103	.
&28	40	_	&68	104	/
&29	41	DOWN CURSOR	&69	105	COPY
&30	48	1	&70	112	ESCAPE
&31	49	2	&71	113	f1
&32	50	D	&72	114	f2
&33	51	R	&73	115	f3
&34	52	6	&74	116	f5
&35	53	U	&75	117	f6
&36	54	0	&76	118	f8
&37	55	P	&77	119	f9
&38	56	[	&78	120	\
&39	57	UP CURSOR	&79	121	RIGHT CURSOR

Bits 0 to 7 refer to the links at the front of the keyboard circuit board on the right-hand side. See OSBYTE &FE for further information about these links.

To convert these internal key numbers to the INKEY numbers they should be EOR (Exclusive ORed) with &FF (255).

Entry parameters: X and Y contain values to be written. Value in X is stored in &ED (old key). Value in Y is stored in &EC (new key).

See also OSBYTE calls with A=&AC and A=&AD.

After call:

- A, X and Y are preserved
- C is undefined

OSBYTE &79 (121)

Keyboard scan

Entry parameters: X determines the key to be detected and also determines the range of keys to be scanned.

Key numbers refer to internal key numbers in the table above.

To scan a particular key:

- X=key number EOR &80 on exit X<0 if the key is pressed.

To scan the matrix starting from a particular key number:

- X=key number of any key pressed or &FF if no key pressed

During the keyboard scan the key whose value is stored in location &EE is ignored. The contents of this location are set to 0 by the operating system on reset.

After call:

- A is preserved
- Y and C are undefined

OSBYTE &7A (122)

Keyboard scan from 16 decimal

No entry parameters

Internal key number (see table above) of the key pressed is returned in X.

This call is directly equivalent to an OSBYTE call with A=&79 and X=16.

After call:

- A is preserved
- Y and C are undefined

OSBYTE &7B (123)

Inform operating system of printer driver going dormant

Entry parameters: X should contain the value 3 (print buffer i.d.)

This OSBYTE call should be used by user printer drivers when they go dormant. The operating system will need to wake up the printer driver if more characters are placed in the printer buffer.

After call:

- A, X and Y are preserved
- C is undefined

OSBYTE &7C (124) *FX 124

Clear ESCAPE condition

No entry parameters

This call clears any ESCAPE condition without any further action. The Tube is informed if active.

The ESCAPE flag is stored as the top bit of location &FF and should never be interfered with directly.

After call:

- A, X and Y are preserved
- C is undefined

OSBYTE &7D (125) *FX 125

Set Escape condition

No entry parameters

This call partially simulates the <ESCAPE> key being pressed. The Tube is informed (if active). An ESCAPE event is not generated.

After call:

- A, X and Y are preserved
- C is undefined

OSBYTE &7E (126) *FX 126

Acknowledge detection of an ESCAPE condition

No entry parameters

This call attempts to clear the ESCAPE condition. All active buffers will be flushed and any open EXEC files closed.

On exit:

- X=&FF if the ESCAPE condition cleared
- X=0 if the ESCAPE condition not cleared

After call:

- A is preserved
- Y and C are undefined

OSBYTE &7F (127)

Check for end-of-file on an opened file

Entry parameters: X contains file handle

On exit:

- X<>0 If end-of-file has been reached
- X=0 If end-of-file has not been reached

After call:

- A and Y are preserved
- C is undefined

OSBYTE &80 (128)

Read ADC channel (ADVAL) or get buffer status

Entry parameters: X determines action and buffer or channel

On entry	On exit
X=0	Y contains channel number (range 1 to 4) showing which channel was last used for ADC conversion. Note that OSBYTE calls with A=&10 (16) and A=&11 (17) set this value to 0. A value of 0 indicates that no conversion has been completed. Bits 0 and 1 of X indicate the status of the two 'fire buttons'.
X=1 to 4	X and Y contain the 16 bit value (X—low, Y—high) read from channel specified by X.
X<0, Y=&FF	If X contains a negative value (in 2's complement notation) then this call will return information about various buffers.
X=255 (&FF)	Keyboard buffer
X=254 (&FE)	RS423 input buffer
X=253 (&FD)	RS423 output buffer
X=252 (&FC)	Printer buffer
X=251 (&FB)	Sound channel 0
X=250 (&FA)	Sound channel 1
X=249 (&F9)	Sound channel 2
X=248 (&F8)	Sound channel 3
X=247 (&F7)	Speech buffer

For input buffers X contains the number of characters in the buffer and for output buffers the number of spaces remaining.

After call:

- A is preserved
- C is undefined

OSBYTE &81 (129)

Read key with time limit (INKEY)

Entry parameters: X and Y specify time limit in centiseconds

If a time limit of n centiseconds is required:

- X=n AND &FF (LSB)
- Y=n DIV &100 (MSB)

Maximum time limit is &7FFF centiseconds (5.5 minutes approx.)

On exit:

- If a character is detected, X=ASCII value of key pressed, Y=0 and C=0.
- If a character is not detected within timeout then Y=&FF and C=1
- If Escape is pressed then Y=&1B (27) and C=1.

If called with Y=&FF and a negative INKEY value in X this call performs a keyboard scan.

On exit, X and Y contain &FF if the key being scanned is pressed.

OSBYTE &82 (130)

Read machine high order address

No entry parameters

This call provides a 16 bit high order address for filing system addresses which require 32 bits. As the BBC microcomputer uses 16 bit addresses internally a padding value must be provided which associates a given address to that machine.

On exit, X and Y contain the padding address (X-high, Y-low) (This address is &FFFF for the BBC microcomputer I/O processor)

After call:

- A is preserved
- C is undefined

OSBYTE &83 (131)

Read top of operating system RAM address (OSHWM)

No entry parameters

On exit, X and Y contain the OSHWM address (X=low byte, Y=high byte)

This call is used by **BASIC** to initialise the value of PAGE.

After call:

- A is preserved
- C is undefined

OSBYTE &84 (132)

Read bottom of display RAM address (HIMEM)

No entry parameters

On exit, X and Y contain the HIMEM address (X=low byte, Y=high byte)

- A is preserved
- C is undefined

OSBYTE &85 (133)

Read bottom of display RAM address for a specified mode

Entry parameters: X determines mode number

On exit, X and Y contain the address (X=low byte, Y=high byte). This call may be used to investigate the consequences of a particular mode's selection.

After call:

- A is preserved
- C is undefined

OSBYTE &86 (134)

Read text cursor position (POS and VPOS)

No entry parameters

On exit:

- X contains horizontal position of the cursor (POS)
- Y contains vertical position of the cursor (VPOS)

After call:

- A is preserved
- C is undefined

OSBYTE &87 (135)

Read character at text cursor position

No entry parameters

On exit:

- X contains character value (0 if char. not recognised)
- Y contains graphics MODE number

After call:

- A is preserved
- C is undefined

OSBYTE &88 (136) *FX 136

*Execute code indirected via USERV (*CODE equivalent)*

This call JSRs to the address contained in the user vector (USERV &200). The X and Y registers are passed on to the user routine.

See the `*CODE` command in Chapter 10.

OSBYTE &89 (137)*FX 137

*Switch cassette relay (*MOTOR equivalent)*

Entry parameters:

- X=0 relay off
- X=1 relay on

The cassette motor LED will reflect the relay state. The cassette filing system calls this routine with Y=0 for write operations and Y=1 for read operations.

After call:

- A is preserved
- X, Y and C are undefined

OSBYTE &8A (138) *FX 138

Insert value into buffer

Entry parameters:

- X identifies the buffer (See OSBYTE call with A=&15/`*FX21` for buffer numbers)
- Y contains the to be value inserted into buffer

After call:

- A is preserved
- C is undefined

OSBYTE &8B (139) *FX 139

*Select file options (*OPT equivalent)*

Entry parameters:

- X contains file option number
- Y contains the option value required

See the `*OPT` command in Chapter 10.

After call:

- A is preserved
- C is undefined

OSBYTE &8C (140) *FX 140

*Select tape filing system (*TAPE equivalent)*

Entry parameters: X selects baud rate

See the **`*TAPE`** command in Chapter 10.

After call:

- A is preserved
- C is undefined

OSBYTE &8D (141) *FX 141

*Select ROM filing system (*ROM equivalent)*

No entry parameters

See the **`*ROM`** command in Chapter 10.

After call:

- A is preserved
- X, Y and C are undefined

OSBYTE &8E (142) *FX 142

Enter language ROM

Entry parameters: X determines which language ROM is entered

The selected language will be re-entered after a soft BREAK. The action of this call is to printout the language name and enter the selected language ROM at &8000 with A=1. Locations &FD and &FE in zero page point to the copyright message in the ROM. When a Tube is present this call will copy the language across to the second processor.

OSBYTE &8F (143) *FX 143

Issue paged ROM service request

Entry parameters: X=service type, Y=argument for service

On exit, Y may contain return argument (if appropriate)

After call:

- A is preserved
- C is undefined

OSBYTE &90 (144) *FX 144

Alter display parameters (*TV equivalent)

Entry parameters:

- X=vertical screen shift in lines
- Y=0 interlace on
- Y=1 interlace off

On exit, X and Y contain the previous settings of the respective parameters

After call:

- A and C are preserved

OSBYTE &91 (145)

Get character from buffer

Entry parameters:

- X contains buffer number (see OSBYTE with A=&15/`*FX21` for buffer numbers)

On exit:

- Y contains the extracted character.
- If the buffer was empty then C=1 otherwise C=0.

After call:

- A is preserved

OSBYTEs &92 to &97 (146 to 151) *FX 146 to 151

Read or Write to mapped I/O

Entry parameters:

- X contains offset within page
- Y contains byte to be written (if write)

OSBYTE call		Memory addressed	Name
read	write		
&92 (146)	&93 (147)	&FC00 to &FCFF	FRED
&94 (148)	&95 (148)	&FD00 to &FDFF	JIM
&96 (150)	&97 (151)	&FE00 to &FEFF	SHEILA

Refer to Chapter 23 for details about the 1 MHz bus.

On exit:

- Read operations return with the value read in the Y register

After call:

- A is preserved
- C is undefined

OSBYTE &98 (152)

Examine Buffer status

Entry parameters: X contains buffer number

For buffer numbers see OSBYTE call with A=&15/`*FX21`

On exit:

- If the buffer is not empty:
  - Y=Pointer to next character to be read from the buffer indexed from zero page locations &FA and &FB.
  - C=0
- If the buffer is empty:
  - Y is preserved
  - C=1

After using this call to examine the next character to be read from a non-empty buffer the instructions `LDA(&FA),Y` will be required. Interrupts should be disabled while the OSBYTE call is made and the buffer examined to prevent any interrupt changing the buffer.

After call:

- A and X are preserved

(This appears to be at variance with an ACORN press release which said that Y was returned containing the value of the character itself.)

OSBYTE &99 (153) *FX 153

Insert character into input buffer, checking for ESCAPE

Entry parameters:

- X contains buffer number:
  - X=0 keyboard buffer
  - X=1 RS423 input
- Y contains the character value

If RS423 input is enabled and X=1 then RS423 ESCAPEs are suppressed (this is the default state plus OSBYTE call with A=&B5 and X=1/`*FX 181,1`), this is identical to OSBYTE call with A=&8A (`*FX 138`).

Otherwise if the character to be inserted is not the ESCAPE character (set by OSBYTE &DC/`*FX 220`) or if ESCAPE characters are to be treated as normal characters (following OSBYTE with A=&E5/`*FX 229`), then an input event (even if input is from RS423) is caused and the character is inserted into the buffer.

If the character is an ESCAPE character and ESCAPEs are not protected (using OSBYTE &C8/`*FX 200`) then an ESCAPE event is generated instead of the keyboard event.

After call:

- A is preserved
- X, Y and C are undefined

OSBYTE &9A (154) *FX 154

Write to video ULA control register and OS copy

Entry parameters: X contains value to be written

This call writes to register 0 of the video ULA and also writes the value in location &248 of the operating system's workspace. For details of the effects of writing to this register see the section on the Video ULA in Chapter 23.

This call also sets the flash counter (stored in location &251) to the mark value (stored in &252).

After call:

- A, X, Y and C are preserved

OSBYTE &9B (155) *FX 155

Write to video ULA palette register and OS copy

Entry parameters: X contains value to be written

This call writes to register 1 of the video ULA and also stores a copy of this value at location &249. The actual value written to the register and the internal copy is X EOR 7. See the section on the Video ULA in Chapter 23.

After call:

- A, X, Y and C are preserved

OSBYTE &9C (156) *FX 156

Read/update 6850 ACIA control register and OS copy

Entry parameters: X and Y determine action

(<register contents> AND Y) EOR X are written to the register

i.e. if Y=&FF and X=0 then no change is made

NOTE: Using this call has no effect on the cassette interface operation and so this is the best way of implementing nonstandard RS423 formats.

On exit, X=old register contents

After call:

- A and Y are preserved
- C is undefined

OSBYTE &9D (157) *FX 157

Fast Tube BPUT

entry parameters: X=byte to be output, Y=file handle

In OS 1.2 this is channelled through the standard BPUT routine. A fast BPUT routine may be implemented in other software.

After call:

- A is preserved
- X, Y and C are undefined

OSBYTE &9E (158)

Read from speech processor

No entry parameters

This call may be used to read data from the serial speech ROM or to read the status register of the speech processor. In order to read from the speech ROM a ***read byte*** command must have previously been sent to the speech processor using OSBYTE call with A=&9F/`*FX 159`. If the speech processor has not been primed in this way then a copy of the speech processor's status register is returned in the Y register.

After call:

- A is preserved
- X and C are undefined

OSBYTE &9F (159) *FX 159

Write to speech processor

Entry parameters: Data/command in Y

This call enables the user to pass opcodes (commands) or bytes of data to the speech processor.

After call:

- A is preserved
- X, Y and C are undefined

OSBYTE &A0 (160)

Read VDU variable value

Entry parameters: X contains the number of the number to be read

On exit, X contains low byte of number and Y contains the high byte

This call reads locations &300,X and &301,X.

After call:

- A is preserved
- C are undefined

OSBYTEs &A6 (166) and &A7 (167)

Read start address of OS variables

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call returns the start address of the memory used by the operating system to store its internal variables.

These values are never written by the operating system except at BREAK and are not read by it either.

After call:

- A is preserved
- X=&90 and Y=&01
- C is undefined

OSBYTEs &A8 (168) and &A9 (169)

Read address of ROM pointer table

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This table of extended vectors consists of 3 byte vectors in the form Location (2 bytes), ROM no. (1 byte). See Chapter 20 for more information on extended vectors.

On exit:

- X=&9F (low byte)
- Y=&0D (high byte)
- i.e. address returned is &0D9F for OS 1.2

After call:

- A is preserved
- C is undefined

OSBYTEs &AA (170) and &AB (171)

Read address of ROM information table

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. the contents of the next location are returned in Y.

This call returns the origin of a 16 byte table, containing one byte per paged ROM. This byte contains the ROM type byte contained in location &8006 of the ROM or contains 0 if a valid ROM is not present. See Chapter 20.

On exit:

- X=&A1
- Y=&02
- i.e. origin address, &02A1 for OS 1.20

After call:

- A is preserved
- C is undefined

OSBYTEs &AC (172) and &AD (173)

Read address of keyboard translation table

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call returns the address of a table which contains the ASCII values of each key where the offset into the table is the internal key number (see OSBYTE with A=&78/`*FX 120`).

Values returned for non-ASCII keys are:

f0	128 (&80)	COPY	139 (&8B)
f1	129 (&81)	LEFT CURSOR	140 (&8C)
f2	130 (&82)	RIGHT CURSOR	141 (&8D)
f3	131 (&83)	DOWN CURSOR	142 (&8E)
f4	132 (&84)	UP CURSOR	143(&8F)
f5	133 (&85)		
f6	134 (&86)	TAB	0
f7	135 (&87)	CAPS LOCK	1
f8	136 (&88)	SHIFT LOCK	2
f9	137 (&89)		
		ESCAPE	27 (&1B)

Non-valid key numbers and **<SHIFT>** or **<CTRL>** return values with no significance.

On exit:

- X=&2B
- Y=&F0
- i.e. address is &F02B for OS 1.20

OSBYTEs &AE (174) and &AF (175)

Read VDU variables origin

NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call returns with the address of the table of internal VDU variables.

On exit:

- x=&00
- Y=&03
- i.e. address is &300 for OS 1.20

OSBYTE &B0 (176)

Read/write CFS timeout counter

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This counter is decremented once every vertical sync pulse (50 times per second) which is also used for OSBYTE &13/`*FX 19`. The timeout counter is used to time interblock gaps and leader tones.

OSBYTE &B1 (177) *FX 177

Read/write input source

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location should contain 0 for keyboard input and 1 for RS423 input (i.e. contains buffer no.)

OSBYTE &B2 (178) *FX 178

Read/write keyboard semaphore

<NEW VALUE> (<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains 0 then keyboard interrupts are ignored. Keyboard interrupts are enabled if it contains &FF.

OSBYTE &B3 (179) *FX 179

Read/write primary OSHWM (for imploded font)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the OSHWM page value for an imploded font (even when character definition RAM explosion has been selected). See OSBYTE &B4/180 and OSBYTE &14/20.

OSBYTE &B4 (180) *FX 180

Read/write OSHWM (equivalent to OSBYTE &83 (131) on read)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location is updated by any character definition RAM explosion which may have been selected and returns with the high byte of the OSHWM address (the low byte always being 0). See OSBYTE &14/20.

OSBYTE &B5 (181) *FX 181

Read/write RS423 mode

<NEW VALUE> =(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains 0 then RS423 input is treated exactly the same as keyboard input i.e. ESCAPEs are recognised, soft keys are expanded and each character entering the input buffer causes a keyboard event.

If this location contains 1 (the usual situation) then ESCAPEs are ignored, soft keys are not expanded and no events are caused.

OSBYTE &B6 (182)

Read character definition explosion state

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the state of font explosion as set by OSBYTE call with A=&14/`*FX 20`.

OSBYTE &B7 (183) *FX 183

Read/write cassette/ROM filing system switch

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains 0 for `*TAPE` selection and 2 for `*ROM` selection. Other values are meaningless.

OSBYTEs &B8 (184) and &B9 (185)

Read video processor ULA registers (OS copies only)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

See OSBYTE calls with A=&9A and A=&9B and Chapter 23 on the Video Hardware.

The last value written to the ULA registers can be read using this method.

These calls should not be used to write to these locations because to do so would make the internal operating system copy of the registers inconsistent with the actual register contents.

OSBYTE &BA (186)

Read ROM number active at last BRK (error)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the ROM number of the paged ROM that was in use at the last BRK.

OSBYTE &BB (187)

*Read number of ROM socket containing **BASIC***

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

BASIC is recognised by the fact that it is a language ROM which does not possess a service entry. This ROM is then selected by the ***BASIC** command (see section 2). If no **BASIC** ROM is present then this location contains &FF. See Paged ROMs.

OSBYTE &BC (188)

Read current ADC channel

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the number of the ADC channel currently being converted. This call should not be used to force ADC conversions, use OSBYTE &11/`*FX 17`.

OSBYTF &BD (189)

Read maximum ADC channel number.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

The maximum channel number to be used for ADC conversions in the range 0 to 4. Set by OSBYTE &16/`*FX 10`.

OSBYTE &BE (190)

Read ADC conversion type, 12 or 8 bits.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

Set to &00, default (12 bit)
Set to &08, 8 bit conversion
Set to &0C, 12 bit conversion

Other values have undefined effects. 8 bit conversion creates values in the same range (0 to &FFFF) but with less precision and two to three times as fast.

OSBYTE &BF (191) *FX 191

Read/write RS423 use flag.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

bit 7 set -	RS423 free.
bit 7 clear -	RS423 busy.
bits 0 to 6 -	undefined.

OSBYTE &C0 (192)

Read RS423 control flag.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is equivalent to OSBYTE &9C/`*FX 156` except that it does not update the 6850 chip. This call should not be used to write the control flag as it would cause the operating system RAM copy to become inconsistent with the 6850 register contents.

OSBYTE &C1 (193) *FX 193

Read/write flash counter.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the number of 1/50th sec. units until the next change of colour for flashing colours.

OSBYTE &C2 (194) *FX 194

Read/write mark period count.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

Equivalent to OSBYTE &09/`*FX 9`.

OSBYTE &C3 (195) *FX 195

Read/write space period count.

<NEW VALUE>=(<old VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

Equivalent to OSBYTE &0A/`*FX 10`.

OSBYTE &C4 (196) *FX 196

Read/write keyboard auto-repeat delay.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is used by OSBYTE &0B/`*FX 11`.

OSBYTE &C5 (197) *FX 197

Read/write keyboard auto-repeat period (rate).

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is used by OSBYTE &0C/`*FX 12`.

OSBYTE &C6 (198) *FX 198

*Read/write *EXEC file handle.*

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains zero if no file handle has been allocated by the operating system.

OSBYTE &C7 (199) *FX 199

*Read/write *SPOOL file handle.*

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the file handle of the current SPOOL file or zero if not currently spooling.

OSBYTE &C8 (200) *FX 200

Read/write ESCAPE, BREAK effect

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

bit 0=0 Normal ESCAPE action
bit 0=1 ESCAPE disabled unless caused by OSBYTE &7D/125
bit 1=0 Normal BREAK action
bit 1=1 Memory cleared on BREAK

e.g. A value 0000001x (binary) will cause memory to be cleared on BREAK.

NOTE: memory location used is: &258

OSBYTE &C9 (201) *FX 201

Read/write keyboard disable.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains 0 then the keyboard is scanned normally otherwise lock keyboard (all keys ignored except <BREAK>).

This call is used by the `*REMOTE` Econet facility.

OSBYTE &CA (202) *FX 202

Read/write keyboard status byte.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

bit 3 - 1 if <SHIFT> is pressed.
bit 4 - 0 if CAPS LOCK is engaged.
bit 5 - 0 if SHIFT LOCK is engaged.
bit 6 - 1 if <CTRL> is pressed.
bit 7 - 1 SHIFT enabled, if a LOCK key is engaged then SHIFT reverses the LOCK.

SHIFT enable (bit 7) may be set by holding <SHIFT> down as the <CAPS LOCK> key is engaged which enables lower-case letters to be typed when capitals are selected by pressing the required key plus <SHIFT>. The only way to set SHIFT enable for the <SHIFT LOCK> key is to use `*FX202,144` (or OSBYTE &CA).

See also OSBYTE with A=&76 (118).

OSBYTE &CB (203) *FX 203

Read/write RS423 handshake extent

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location determines the space remaining in the RS423 input buffer when RS423 input is halted by the operating system entering a buffer full state (which sets the RTS line high). The default value is 9. The free space remaining in the buffer allows some manipulation of the buffer contents to be carried out before being passed on. The value selected should reflect the response time at the transmission end and the time taken for the operating system to act upon the buffer full situation.

OSBYTE &CC (204) *FX 204

Read/write RS423 input suppression flag.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains 0 then RS423 input is accepted otherwise RS423 input is ignored (RS423 receive errors will still cause an event).

OSBYTE &CD (205) *FX 205

Read/write cassette/RS423 selection flag.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

- If this location contains 0 then RS423 data is channelled to the RS423 hardware.
- If this location contains &40 then RS423 data is channelled to the cassette hardware.

- This location is only checked, and so any change will only come into effect, when a baud rate selection is made using `*FX 7` or `8`.

OSBYTE &CE (206) *FX 206

Read/write Econet OS call interception status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If bit 7 of this location is set then all OSBYTE and OSWORD calls (except those sent to paged ROMs) are indirected through the Econet vector (&224) to the Econet. Bits 0 to 6 are ignored.

OSBYTE &CF (207) *FX 207

Read/write Econet read character interception status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If bit 7 of this location is set then input is pulled from the Econet vector.

OSBYTE &D0 (208) *FX 208

Read/write Econet write character interception status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If bit 7 of this location is set then output is directed to the Econet. Output may go through the normal write character on return from the Econet code.

OSBYTE &D1 (209) *FX 209

Read/write speech suppression status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the value sent to the speech processor when speech is output. A value of &50 represents the SPEAK op. code and is the default value (speech enabled). Writing &20 (NOP) to this location will disable speech.

OSBYTE &D2 (210) *FX 210

Read/write sound suppression status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains any value other than 0 then sound output is disabled.

OSBYTE &D3 (211) *FX 211

Read/write BELL (CTRL G) channel.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the channel number to be used for the BELL sound. Default value is 3.

OSBYTE &D4 (212) *FX 212

Read/write BELL (CTRL G) SOUND information.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains a byte which determines either the amplitude or the ENVELOPE number to be used by the BELL sound. If an ENVELOPE is specified then the value should be set to (ENVELOPE no.-1)*8. Similarly an amplitude in the range -15 to 0 must be translated by subtracting 1 and multiplying by 8.

The least significant three bits of this location contain the H and S parameters of the SOUND command (see Chapter 5).

e.g. Try `*FX 212,216` for a softer BELL sound (amplitude -4).

Default value 144 (&90).

OSBYTE &D5 (213) *FX 213

Read/write bell (CTRL G) frequency.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This value contains the pitch parameter (as used by SOUND command third parameter) used for the BELL sound.

Default value 101 (&65).

OSBYTE &D6 (214) *FX 214

Read/write bell (CTRL G) duration.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next locationare returned in Y.

This value contains the duration parameter (as for SOUND command) used for the BELL sound.

Default value 7.

OSBYTE &D7 (215) *FX 215

Read/write start up message suppression and !BOOT option status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

bit 7 If clear then ignore OS startup message.
If set then print up OS startup message as normal.

bit 0 If set then if an error occurs in a !BOOT file in `*ROM` carry on but if an error is encountered from a disc !BOOT file because no language has been initialised the machine locks up.
If clear then the opposite will occur, i.e. locks up if there is an error in `*ROM`

This can only be over-ridden by a paged ROM on initialisation or by intercepting BREAK, see OSBYTE calls &F7 to &F9.

OSBYTE &D8 (216) *FX 216

Read/write length of soft key string.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the number of characters yet to be read from the soft key buffer of the current soft key. It may be useful to set this value to 0 to cancel a soft key expansion without clearing the input buffer. To clear input buffer use OSBYTE &0F/`*FX 15`.

OSBYTE &D9 (217) *FX 217

Read/write number of lines since last halt in page mode.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the number of lines printed since the last page halt.

OSBYTE &DA (218) *FX 218

Read/write number of items in the VDU queue.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This contains the 2's complement negative number of bytes still required for the execution of a VDU command.

Writing 0 to this location can be a useful way of abandoning a VDU queue otherwise writing to this location is not recommended.

OSBYTE &DB (219) *FX 219

Read/write character value returned by pressing TAB key.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the value to be returned by the **<TAB>** key. It is possible to use the **<TAB>** key as a soft key by setting this location to &80+n where n is the soft key number.

Default value is 9 (forward cursor 1 char.).

OSBYTE &DC (220) *FX 220

Read/write Escape character.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains the ASCII character (and key) which will generate an ESCAPE.

e.g. `*FX 220,32` will make the **<SPACE>** bar the **<ESCAPE>** key.

Default value &1B (27).

OSBYTEs &DD (221) to &EO (224) *FX 221 to 224

Read/write I/P buffer code interpretation status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

These locations determine the effect of the character values &C0 (192) to &FF (255) when placed in the input buffer. See OSBYTEs &E1 (225) to &E4 (228) for details about the different effects which may be selected. Note that these values cannot be inserted into the input buffer from the keyboard. RS423 input or a user keyboard handling routine may place these values into the input buffer.

- OSBYTE &DD affects interpretation of values &C0 to &BF
- OSBYTE &DE affects interpretation of values &D0 to &CF
- OSBYTE &DF affects interpretation of values &E0 to &EF
- OSBYTE &EO affects interpretation of values &F0 to &FF

Default values &01,&D0,&E0 and &F0 (respectively)

OSBYTE &E1 (225) *FX 225

Read/write function key status (soft keys or codes).

Input buffer characters &80 to &8F.

OSBYTE &E2 (226) *FX 226

Read/write SHIFT+function key status (soft key or code).

Input buffer characters &90 to &9F.

OSBYTE &E3 (227) *FX 227

Read/write CTRL+ function key status (soft key or code).

Input buffer characters &A0 to &AF.

OSBYTE &E4 (228) *FX 228

Read/write CTRL+SHIFT+function key Status (soft key or code).

Input buffer characters &B0 to &BF.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

These locations determine the action taken by the operating system when a function key is pressed.

value 0 - totally ignore key.
value 1 - expand as normal soft key.
value 2 to &FF - add n (base) to soft key number to provide 'ASCII' code.

The default settings are:

fn keys alone	&01	expand using soft key strings
fn keys+SHIFT	&80	code &80-soft key number
fn keys+CTRL	&90	code &90-soft key number
fn keys+SHIFT+CTRL	&A0	code &A0-soft key number

When the <BREAK> key is pressed a character of value &CA is entered into the input buffer. The effect of this character may be set independently of the other soft keys using OSBYTE &DD (221). One of the other effects of pressing the <BREAK> key is to reset this OSBYTE call and so the usefulness of this facility is limited.

OSBYTE &E5 (229) *FX 229

Read/write status of ESCAPE key (escape action or ASCII code).

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains 0 then the <ESCAPE> key has its normal action. Otherwise treat currently selected <ESCAPE> key as an ASCII code.

OSBYTE &F6 (230) *FX 230

Read/write flags determining ESCAPE effects.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains 0 then when an ESCAPE is acknowledged (using OSBYTE &7E/*FX 126) then:

- ESCAPE is cleared
- EXEC file is closed (if open)
- Purge all buffers (including input buffer)
- Reset VDU paging counter.

If this location contains any value other than 0 then ESCAPE causes none of these.

OSBYTE &E7 (231) *FX 231

Read/write IRQ bit mask for the user 6522.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR N

The old value is returned in X. The contents of the next location are returned in Y.

See the section on the User VIA in Chapter 23.

Default value &FF.

OSBYTE &E8 (232) *FX 232

Read/write IRQ bit mask for 6850 (RS423).

<NEW VALUE>=(<OLD VALUE> AND Y) EOR N

The old value is returned in X. The contents of the next location are returned in Y.

See the serial interface in Chapter 23.

Default value &FF.

OSBYTE &E9 (233) *FX 233

Read/write interrupt bit mask for the system 6522.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR N

The old value is returned in X. The contents of the next location are returned in Y.

See the section on the system 6522 in Chapter 23.

Default value &FF.

OSBYTE &EA (234)

Read flag indicating Tube presence.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains 0 if a Tube system is not present and &FF if Tube chips and software are installed. No other values are meaningful or valid.

OSBYTE &EB (235)

Read flag indicating speech processor presence.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains 0 if the speech processor is not present and &FF if it is.

OSBYTE &EC (236) *FX 236

Read/write write character destination status.

<NEW VALUE>=(<OLD VALUE> AND Y EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is used by OSBYTE &3/`*FX 3`.

OSBYTE &ED (237) *FX 237

Read/write cursor editing status.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is used by OSBYTE &4/`*FX 4`.

OSBYTEs A=&EE (238), &EF (239) and &F0 (240)

Read/write location &27E, &27F and &280.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

These locations are not used by the operating system. Default values 0.

OSBYTE &F1 (241) *FX 241

Read/write location &281.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is not used by the operating system and is unlikely to be used by later issues either. This location is reserved as a user flag for use with `*FX 1`.

Default value 0.

OSBYTE &F2 (242)

Read copy of the serial processor ULA register.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

See the section on the serial interface in Chapter 23 for the significance of this register. This call should not be used for writing as it would place this copy of the register contents out of sync with the register contents themselves.

All the serial ULA functions can be controlled with:

- OSBYTE with A=&89/*FX 137 motor control
- OSBYTE with A=&CD/*FX 205 cassette/RS423 select
- OSBYTE with A=&7,&8/*FX 7,8 RS423 baud rate control

OSBYTE &F3 (243)

Read timer switch state.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

The operating system maintains two internal clocks which are updated alternately. As the operating system alternates between the two clocks it toggles this location between values of 5 and 10. These values represent offsets from &28D where the clock values are stored.

OSBYTE &F4 (244) *FX 244

Read/write soft key consistency flag.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

If this location contains 0 then the soft key buffer is in a consistent state. A value other than 0 indicates that the soft key buffer is in an inconsistent state (the operating system does this during soft key string entries and deletions). If the soft keys are in an inconsistent state during a soft break then the soft key buffer is cleared (otherwise it is preserved).

OSBYTE &F5 (245) *FX 245

Read/write printer destination flag.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is used by OSBYTE &5/`*FX 5`. Using this call does not check for the printer previously selected being inactive or inform the user printer routine. See Expansion vectors, Chapter 16.

OSBYTE &F6 (246) *FX 246

Read/write character ignored by printer.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is used by OSBYTE &6/`*FX 6`.

OSBYTEs &F7 (247), &F8 (248) and &F9 (249)

Read/write BREAK intercept code.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

The contents of these locations must be JMP instruction for BREAKs to be intercepted (the operating system identifies the presence of an intercept by testing location &287 contents equal to &4C -JMP). This code is entered twice during each break. On the first occasion C=0 and is performed before the reset message is printed or the Tube initialised. The second call is made with C=1 after the reset message has been printed and the Tube initialised.

NOTE: Memory locations used in OS1.20:

Jmp	lobyte	hibyte
&287	&288	&289

OSBYTEs &FA (250) and &FB (251) *FX 250 to 251

Read/write locations &28A and &28B.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

Not used by the operating system.

Default values 0.

OSBYTE &FC (252) *FX 252

Read/write current language ROM number.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location is set after use of OSBYTE &8E/`*FX 126`. This ROM is entered following a soft BREAK or a BRK (error).

OSBYTE &FD (253)

Read hard/soft BREAK.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains a value indicating the type of the last BREAK performed.

value 0- soft BREAK
value 1 - power up reset
value 2- hard BREAK

OSBYTE &FE (254) *FX 254

Read/write available RAM.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location contains a value indicating the available RAM.

value &40 — 16 K (usually model A)
value &80 — 32 K (usually model B)

OSBYTE &FF (255) *FX 255

Read/write start up options.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This location is determined by the 8 links on the right hand front corner of the keyboard PCB following a hard BREAK.

bits 0 to 2	screen MODE selected following reset. (MODE number = 3 bit value)
bit 3	if clear reverse action of <SHIFT> <BREAK>
bits 4 and 5	used to set disc drive timings (see below).
bits 6 and 7	not used by operating system. (reserved for future applications)

The disc drive timing links govern the speed and other timings when reading and writing floppy discs. These values vary, depending on the type of disc interface and controller chip you have fitted. The common Acorn types are shown below, other 3rd party systems may vary, refer to the handbook that came with your system to make sure.

Disc drive timing links for the 8271 and 1770/2 chips:

link 3	link 4	step time i8271	step time WD1770	step time WD1772	settle time	head load
1	1	4	6	6	16	0
1	0	6	12	12	16	0
0	1	6	20	2	50	32
0	0	24	30	3	20	64

15. OSWORD calls

The OSWORD routines are a similar concept to the OSBYTE routines except that instead of the parameters being passed in the X and Y registers, parameters are placed in a parameter block, the address of which is sent to the OSWORD routine in the X (low byte) and Y (high byte) registers.

OSWORD - OS call specified by the contents of A taking parameters in a parameter block.

Call address &FFF1

Indirected through &20C

On entry:

- A selects an OSWORD routine.
- X contains low byte of the parameter block address.
- Y contains high byte of the parameter block address.

OSWORD calls which are called with accumulator values in the range &E0 (224) to &FF (255) are passed to the USERV (&200). The routine indirected through the USERV is entered with the register contents unchanged from the original OSWORD call. (See Vectors section, and Operating system commands section 2 for more information about the user vector.)

Other unrecognised OSWORD calls are offered to the paged ROMs (see Chapter 20 on Paged ROMs, reason code 8).

OSWORD summary	
A=0	Read line from currently selected input into memory.
A=1	Read system clock.
A=2	Write system clock.
A=3	Read interval timer.
A=4	Write interval timer.
A=5	Read byte of I/O processor memory.
A=6	Write byte of I/O processor memory.
A=7	Perform a SOUND command.
A=8	Define an ENVELOPE.
A=9	Read pixel value.
A=&A	Read character definition.
A=&B	Read palette value for a given logical colour.
A=&C	Write palette value for a given logical colour.
A=&D	Read previous and current graphics cursor positions.

OSWORD call with A=&0

Read line from input

This routine takes a specified number of characters from the currently selected input stream. Input is terminated following a **<RETURN>** or an **<ESCAPE>**. **<DELETE>** (&7F/127) deletes the previous character and **<CTRL>** **<U>** (&15/21) deletes the entire line.

If characters are presented after the maximum line length has been reached the characters are ignored and a BEL (ASCII 7) character is output.

The parameter block:

XY+ 0	Buffer address for input	LSB
1		MSB
2	Maximum line length	
3	Minimum acceptable ASCII value	
4	Maximum acceptable ASCII value	

Only characters greater or equal to XY+3 and lesser or equal to XY+4 will be accepted.

On exit:

- C=0 if a carriage return terminated input.
- C=1 if an ESCAPE condition terminated input.
- Y contains line length, including carriage return if used.

OSWORD call with A=&1

Read system clock

This routine may be used to read the system clock (used for the TIME function in **BASIC**). The five byte clock value is written to the address contained in the X and Y registers. This clock is incremented every hundredth of a second and is set to 0 by a hard BREAK.

OSWORD call with A=&2

Write system clock

This routine may be used to set the system clock to a five byte value contained in memory at the address contained in the X and Y registers.

OSWORD call with A=&3

Read interval timer

This routine may be used to read the interval timer (Used for events). The five byte clock value is written to the address contained in the X and Y registers.

OSWORD call with A=&4

Write interval timer

This routine may be used to set the interval timer to a five byte value contained in memory at the address in the X and Y registers.

OSWORD call with A=&5

Read I/O processor memory

A byte of I/O processor memory may be read across the Tube using this call. A 32 bit address should be contained in memory at the address contained in the X and Y registers.

XY+ 0	LSB of address to be read
1	
2	
3	MSB of address to be read

If the I/O processor uses 16 bit memory addressing only least significant two bytes need to be specified.

On exit:

- The byte read will be contained in location XY+4.

OSWORD call with A=&6

Write I/O processor memory

This call permits I/O processor memory to be written across the Tube. A 32 bit address is contained in the parameter block addressed by the X and Y registers and the byte to be written should be placed in XY+4.

OSWORD call with A=&7

SOUND command

This routine takes an 8 byte parameter block addressed by the X and Y registers. The 8 bytes of the parameter block may be considered as the four parameters used for the `SOUND` command in **BASIC**, see chapter 5.

e.g. To perform a `SOUND 1,-15,200,20`

XY+					
XY+	0	Channel	LSB	1	&01
	1		MSB		&00
	2	Amplitude	LSB	-15	&F1
	3		MSB		&FF
	4	Pitch	LSB	200	&C8
	5		MSB		&00
	6	Duration	LSB	20	&14
	7		MSB		&00

This call has exactly the same effect as the `SOUND` command.

OSWORD call with A=&8

Define an ENVELOPE

The ENVELOPE parameter block should contain 14 bytes of data which correspond to the 14 parameters described in the `ENVELOPE` command. This call should be entered with the parameter block address contained in the X and Y registers. See Chapter 5 for a detailed description on Envelopes.

OSWORD call with A=&9

Read pixel value

This routine returns the status of a screen pixel at a given pair of X and Y co-ordinates. A four byte parameter block is required and result is contained in a fifth byte.

XY+	
XY+	0	LSB of the X co-ordinate
	1	MSB of the X co-ordinate
	2	LSB of the Y co-ordinate
	3	MSB of the Y co-ordinate

On exit:

- XY+4 contains the logical colour at the point or &FF if the point specified was off screen.

OSWORD call with A=&A

Read character definition

The 8 bytes which define the 8 by 8 matrix of each character which can be displayed on the screen may be read using this call. The ASCII value of the character definition to be read should be placed in memory at the address stored in the X and Y registers. After the call the 8 byte definition is contained in the following 8 bytes.

XY+ 0	Character required
1	Top row of character definition
2	Second row of character definition
.	.
.	.
8	Bottom row of character definition

OSWORD call with A=&B

Read palette

The physical colour associated with each logical colour may be read using this routine. On entry the logical colour is placed in the location at XY and the call returns with 4 bytes stored in the following four locations corresponding to a `VDU 19` statement.

e.g. Assuming that a `VDU 19,1,3,0,0,0` had previously been issued then OSWORD &B with 1 at XY would yield:

XY+ 0	1	logical colour
1	3	physical colour
2	0	padding for future expansion
3	0	
4	0	

OSWORD call with A=&C

Write palette

This call performs the same task as a `VDU 19` command (which can be used from machine code using OSWRCH). The advantage of using this OSWORD call rather than the conventional VDU route is that there is a significant saving in time. Another advantage is that OSWORD calls can be used in interrupt routines while VDU routines cannot. This call works in the same way as OSWORD &B (see above); a parameter block should be set up with the logical colour being defined at XY, the physical colour being assigned to it in XY+1 and XY+2 to XY+4 containing padding 0s.

OSWORD call with A=&D

Read last two graphics cursor positions

The operating system keeps a record of the last two graphics cursor positions in order to perform triangle filling if requested. These cursor positions may be read using this call. X and Y should provide the address of 8 bytes of memory into which the data may be written.

XY+ 0	previous X co-ordinate	low byte
1	previous X co-ordinate	high byte
2	previous Y co-ordinate	low byte
3	previous Y co-ordinate	high byte
4	current X co-ordinate	low byte
5	current X co-ordinate	high byte
6	current Y co-ordinate	low byte
7	current Y co-ordinate	high byte

16. Vectors

One of the features of the BBC microcomputer that greatly enhances its power is the extensive use of VECTORS. A vector is a word in memory containing the address of a service routine. Many of the more important operating system routines are indirected through vectors. The write character routine, OSWRCH, uses a vector called WRCHV. Each time OSWRCH is called it jumps to the routine whose address is contained in WRCHV. All of the vectors used for operating system routines are initialised on reset by the operating system. Normally each vector contains the address of the relevant routine in the operating system.

The advantage of using vectors is that standard operating system routines can be intercepted by changing the address contained in the vector. The user can replace the address in the vector with the address of his own routine. This routine may totally replace the operating system routine or it may perform some function and then pass control to the routine whose address was previously contained in the vector (when changing the operating system vector the user would have to save the old contents of the vector and store them in a vector of his own in his routine).

Some of the internal operating system routines are also indirected through vectors. This enables the user to alter the function of the operating system in certain areas. For example, the buffer management routines can be modified by intercepting INSV and REMV which indirect the buffer insertion and removal routines.

Vectors also provide a way of allowing the user to enter routines for which there is no direct entry point such as the Filing System Control vector.

When providing new routines for vectors, the programmer should note that there are two **BASIC** strategies for returning from vectored code, these are:

a) The routine provided completely replaces the standard code. In this case, the routine should exit with an RTS instruction (or an RTI if it is one of the interrupt or break vectors).
b) The routine provided does not completely replace the standard code. This is the more usual case, and routines of this sort should exit with a JMP (oldvec), where oldvec is the old vector contents. Routines passing control

on to further code should exit with the registers preserved from the entry to the routine.

c) A combination of the above. This can occur with vectors that provide more than one function (this is most of them), and some of the functions are to be completely replaced, and others passed on to the standard routine. In this case, both of the above exit strategies should be adopted.

The use of strategy (b) allows a whole series of routines to be daisy chained together on one vector, each taking over one or two functions of that vector. Wherever possible, this strategy should be used, as it maximises the possible flexibility of the system.

The main operating system vectors reside in page two of memory. There is also an extended vector space in page &D for use by paged ROMs (see the section on vectored entry to paged ROMs in Chapter 20 for more details).

The operating system vectors are:

&200,1	**USERV**. The user vector. This is used to take out certain unrecognised operating system calls, including instructions *CODE and *LINE.
&202,3	**BRKV**. The break vector. This is used to trap errors within the system. This vector is normally set by the current language.
&204,5	**IRQ1V**. The primary interrupt vector. All interrupts are directed through this vector.
&206,7	**IRQ2V**. The secondary interrupt vector. All interrupts unrecognised by the standard interrupt processing routine, or which have been masked out are passed through this vector.
&208,9	**CLIV**. The command line interpreter vector. All operating system commands (*commands) are passed through this vector.
&20A,B	**BYTEV**. The OSBYTE indirection vector. All OSBYTE calls are passed through this vector.
&20C,D	**WORDV**. The OSWORD indirection vector. All OSWORD calls are passed through this vector.
&20E,F	**WRCHV**. Write character vector. All writes to the screen and printer etc. are passed through this vector.
&210,1	**RDCHV**. Read character vector. All reads from the currently selected input stream are passed through this vector.
&212,3	**FILEV**. Read/write a whole file. All file loads and saves are passed through this vector.

&214,5	**ARGSV**. Read/write file arguments. All calls of OSARGS pass through this vector.
&216,7	**BGETV**. Read one byte from a file. Used for **BASIC** BGET and `*EXEC` commands.
&218,9	**BPUTV**. Put one byte to a file. Used for **BASIC** BPUT and `*SPOOL` commands.
&21A,B	**GBPBV**. Get/put a block of bytes from/to a file.
&21C,D	**FINDV**. Open/close a file for BPUT,BGET,GBPB,ARGS calls.
&21E,F	**FSCV**. Various filing system control functions.
&220,1	**EVENTV**. Pointer to the event handling routine.
&222,3	**UPTV**. Pointer to user print routine.
&224,5	**NETV**. Used by Econet to take control of the computer.
&226,7	**VDUV**. Used by the VDU driver to direct unrecognised `VDU 23` and `PLOT` commands.
&228,9	**KEYV**. Used by the operating system for all keyboard access. Enables the use of external keyboards.
&22A,B	**INSV**. Insert into buffer vector.
&22C,D	**REMV**. Remove from buffer vector.
&22E,F	**CNPV**. Count / purge buffer vector.
&230,1	**IND1V**. Spare vector.
&232,3	**IND2V**. Spare vector.
&234,5	**IND3V**. Spare vector.

These vectors are now described in more detail.

The user vector, USERV &200,1

The user vector exists to allow the user some expansion facilities without needing to take full control of one of the main vectors. Two operating system commands, `*CODE` and `*LINE` are passed through the user vector, as are 32 OSWORD calls.

When a routine which is pointed to by the user vector is entered, the contents of the accumulator describe the type of entry required:

A=0	*CODE has been executed (or OSBYTE &88), X and Y contain the two parameters. See OS commands, Chapter 10.
A=1	*LINE has been executed. X and Y point to the rest of the command line. See OS commands, Chapter 10.
A=&E0...&FF	OSWORD has been entered with the accumulator in the range &EO...&FF. Registers as on entry to OSWORD.

The break vector, BRKV &202,3

The break vector is primarily used to trap errors. On entry to the break vector the following conditions prevail:

a) The registers A, X and Y are unchanged from when the BRK instruction was executed.

b) The stack is prepared ready for an RTI instruction to return to the instruction following the BRK instruction. For this purpose, the BRK instruction is taken as a two byte instruction, i.e. the instruction pointed to is two bytes after the BRK instruction. This is because many of the 6502 instructions that might be replaced by a BRK instruction are two bytes long.

c) Locations &FD and &FE contain the address of the byte after the BRK instruction, which normally contains the error number. See below.

Note that although a fully prepared exit from a BRK instruction is possible, neither the operating system or **BASIC** expect a return from this vector. Possibly fatal results may occur if such a return is made as paged ROM software typically stores the BRK, error number and message in page one below the stack, returning there is very hazardous. The exception to this is when using the BRK instruction as a breakpoint in user supplied machine code, and is not used as a standard error generating mechanism.

Note also that the break vector only refers to the BRK assembler instruction, it should not be confused with the **<BREAK>** key on the keyboard, which causes a hardware reset. This vector is changed to its default state during the course of processing such a reset.

The BBC microcomputer adopts a standard pattern of bytes following a BRK instruction, this is:

- A single byte error number
- An error message
- A zero byte to terminate the message

The interrupt vectors, IRQ1V and IRQ2V &204-7

For the entry conditions to these vectors, refer to Chapter 19 on interrupts.

Main vector zone, vectors from &208-21F

Entry conditions for these vectors are covered in the following Chapters:

- Chapter 14 - 'OSBYTE calls' for BYTEV,
- Chapter 15 - 'OSWORD calls' for WORDV,
- Chapter 10 - 'Operating System calls' for WRCHV and RDCHV,
- Chapter 21 - 'Filing systems' for FILEV to FSCV.

The event vector, EVNTV &220,1

For entry conditions to the event vector refer to Chapter 18 - Events.

The user print vector, UPTV &222,3

This vector contains the address of the user provided printer driver. The facility for providing a user print driver is important, since not all printers have Centronics parallel, or standard serial interfaces. Using this vector allows specialised code for driving the hardware on a particular printer to be inserted.

The user printer driver acts as an interface between the printer buffer in memory and the printer hardware. The driver is responsible for removing the characters from the printer buffer when the printer is ready to accept them, and passing them on to the printer hardware. The printer driver is regularly entered, allowing it to

poll its hardware and the printer buffer, and informing it of relevant changes in printer conditions.

The user printer driver should preserve all registers. On entry, the X register contains the buffer number that should be 'tapped' to claim bytes for output. The Y register contains the printer type selected by `*FX 5`. The driver should recognise calls to itself by checking the Y register against the printer type it recognises. The normal number to recognise is 3, but it is possible to design a printer driver that will run two types of printer. The driver can also use any number from 5 to 255, 4 being reserved for the networked printer. If the user printer driver recognises a call to itself, it should respond to the following codes in the accumulator:

A=0 The operating system enters the user print routine on interrupt once every 10 milliseconds, if it is not dormant. The user print routine can poll the printer in response to this call. This should eliminate the need for the user printer hardware to generate interrupts.

A=1 The printer is to be activated because at least one character is in the print buffer. This entry is only made if the driver had previously marked itself as dormant with OSBYTE &7F. The driver should take one character from the designated buffer and send it to the printer. It should exit with the carry flag clear if the printer is going active. When the printer is active it is no longer warned of characters entering the buffer, but is expected to use the 10ms (A=0) entry to continually poll the printer hardware and the designated buffer until the last character in the buffer has been printed.

A=2 Warning that a `VDU 2` has been received. Note that characters can be printed without a `VDU 2` occurring if some options of `*FX 3` are used.

A=3 Warning that a `VDU 3` has been received.

A=5 Printer type change. The X register contains the new printer type being selected. This call is made every time an OSBYTE 5 is made, irrespective of the currently selected printer type, or that being selected.

Printer drivers should declare themselves inactive (with OSBYTE &7B) after they have finished printing the last character in the buffer. This has the following effects:

a) The user is allowed to select a new printer type with OSBYTE 5.
b) The operating system no longer offers interrupts to the user print driver.
c) The printer driver will be warned with an entry code 1 when a new character is printed.

The Econet vector, NETV &224,5

The net vector is used for various network effects. In non networked machines, this vector can be used for a variety of purposes. The user can be entirely disconnected from the operating system with this call, or have all his actions vetted.

This vector has rather more program protection applications than main line uses. Usually other vectors which allow more specific control of functions would be used.

the effects are specified by the value in the accumulator. The net routine should preserve registers. The net effect codes are:

0,1,2,3,5 These codes are used to control the networked printer. The printer is in every respect the same as a user printer, see the previous section on the user print driver. Printer type number 4 is nominally allocated to the networked printer.

4 Write character attempted. This call to the net vector is only made when enabled by OSBYTE &D0. On entry Y is the character to be output. On exit, if the carry flag is set the output of the character is not passed on to the operating system.

6 Read character attempted. This call to the net vector is only made when enabled by OSBYTE &CF. The net system must provide a character for processing on exit in the accumulator.

7 OSBYTE attempted. This call to the net vector is only made if enabled with OSBYTE &CE. The entry parameters of the OSBYTE call are held in &EF, &F0, and &F1 for A,X and Y registers respectively. If on exit the overflow flag is set, the user is prevented from making the call.

8 OSWORD attempted. Exactly as call 7 (OSBYTE), only an OSWORD call was attempted.

&0D A line has been entered with OSWORD 1, and is now complete. This is a warning to the net system that it can now take over the read character input without too much mess.

The VDU extension vector, VDUV &226,7

This vector is used whenever an unrecognised VDU command occurs. This happens in three situations:

1) `VDU 23,n` has been issued with `n` in the range 2. .31. The vector is entered with the carry flag set. The accumulator contains `n`. Locations &31C. .&323 contain the eight parameters always sent with the `VDU 23` command.
2) A `PLOT` command has been issued in a non-graphics mode.
3) An unrecognised PLOT number has been used.

In cases (2) and (3), the vector is entered with the carry flag clear. The accumulator contains the PLOT number. Locations &320 to &323 contain the X and Y co-ordinates sent via the `PLOT` command. If the command was issued within a graphics mode, the co-ordinates are converted to internal co-ordinates, with relative plots and the graphics origin taken into account.

See the memory usage section for more information on internal co-ordinates and VDU variables space layout.

The keyboard control vector, KEYV &228,9

This vector is used whenever the keyboard is accessed. It has the following entry conditions:

C=0,V=0: Test the SHIFT and CTRL keys, exit with the N (minus) flag set if the CTRL key is pressed, and the V flag (overflow) is set if the SHIFT key is pressed.

C=1,V=0: Scan the keyboard. Exactly as OSBYTE call &79. On exit, the accumulator is equal to the X register on exit.

C=0,V=1: Key pressed interrupt entry. Each time a key is pressed the system VIA generates an interrupt. The operating system uses this interrupt to provide the 'type ahead' facility.

C=1,V=1: Timer interrupt entry. This entry is used for most of the keyboard processing. Keyboard auto repeat timing is performed during this call, as is the two key rollover processing.

This vector can usefully be used as an entry point into the operating system, as well as replacing the normal routine provided. This entry can be used, for example, to test the control and shift keys.

The buffer insert vector, INSV &22A,B

The routine indirected through this vector is used by the operating system to enter a character into a buffer.

On entry:

- A = character to be inserted.
- X = buffer number. No range checking is done on this number.

On exit:

- A,X preserved.
- Y is undefined.

- C is set if the insertion failed due to the buffer being full. It is the responsibility of the calling routine to attempt retries, or abandon the insertion, if the insertion failed.

The buffer remove vector, REMV &22C,D

The routine indirected through this vector is used by the operating system to remove a character from a buffer, or to examine the buffer only.

On entry:

- X= buffer number. No range checking is done on this number.
- The overflow flag is set if only an examination is needed.

If the buffer is only examined, the next character to be withdrawn from the buffer is returned, but not removed, hence no buffer empty event can be caused.

If the last character is removed, a buffer empty event will be caused.

On exit:

- A is the next character to be removed, for the examine option, undefined otherwise.
- X is preserved.
- Y is the character removed for the remove option.
- C is set if the buffer was empty on entry.

The buffer count/purge vector, CNPV &22E,F

The routine indirected through this vector is used by the operating system to count the entries in a buffer or to purge the contents of a buffer.

On entry:

- X= buffer number. No range checking is done on this number.
- The overflow flag is set if the buffer is to be purged.
- The overflow flag is clear if the buffer is to be counted.

For a count operation, if the carry flag is set, the amount of space left in the buffer is returned, otherwise the number of entries in the buffer is returned.

On exit:

- For purge:
  - X and Y are preserved.
- For count:
  - X=low byte of result Y=high byte of result
- A is undefined.
- V,C are preserved.

The spare vectors, IND1V...IND3V &230. .235

These vectors are not used by OS 1.20. Future versions of the operating system may use these vectors or Acorn may allocate them for use by application software.

The default vector table

There exists within the operating system (OS 1.2 onwards), a lookup table of the default vector contents. This table is useful for restoring the vectors after changing them, or for software protection (some people use events to get past software protection). The information on the table is given thus:

Location:

- &FFB6 length of lookup table in bytes.
- &FFB7 low byte of the address of the table.
- &FFB8 high byte of the address of the table.

17. Memory usage

This chapter describes how the memory in the BBC microcomputer is allocated between the different contenders. The area allocated to the operating system is described in some detail, but that used by the language and user programs is only outlined, as it varies from language to language.

Some of the information in this section is also to be found elsewhere (see Chapters 21 on filing systems and 20 on paged ROMs). The majority of this information is specific to OS 1.20, although most of it is correct on other series 1 operating systems, and the general overview is true for operating system 0.1.

It should be noted that all locations described here are highly 'unofficial' and are not documented by Acorn. For compatibility with future operating systems, users should not use any of these locations directly unless it is totally unavoidable. Access to these locations via OSBYTE calls should remain fairly operating system independent. Acorn have not documented OSBYTE &A0, so the VDU variable locations should therefore not be relied upon to remain constant. The locations listed here should prove of great use to those disassembling the operating system ROM.

Zero page

The zero page on the 6502 is very valuable, as many instructions and addressing modes need to work through page zero. For this reason, areas of zero page are allocated to each of the main memory contenders.

Zero page is allocated thus:

&00-&8F are allocated to the current language. **BASIC**' reserves locations &70-&8F for the user.

&90-&9F are allocated to the Econet system.

&A0-&A7 are allocated to the current NMI owner (see the relevant section in Chapter 20, paged ROMs). This area is not used on **BASIC** cassette machines. It is used extensively by the disc and network filing systems.

&A8-&AF are allocated for use by operating system commands during execution.

&B0-&BF are allocated as filing system scratch space. but are not exclusively used by the currently active filing system.

&C0-&CF are allocated to the currently active filing system. This area is nominally private, and will not be altered unless the filing system is changed, or the absolute workspace is claimed (see Chapter 20, paged ROMs).

&D0-&E1 are allocated to the VDU driver.

&D0 is the VDU status as returned by OSBYTE &75.

&D1 contains a byte mask for the current graphics point. This byte indicates which bits in the screen memory byte correspond to the point. For example, for the rightmost pixel in a two colour mode, this byte would contain &01, and for a sixteen colour mode, &55.

&D2 and **&D3** are the text colour bytes to be ORed and EORed into memory, respectively. When writing text to the screen in modes 0 to 6, the pattern byte to be written to the screen is first ORed with the contents of &D2, and then EORed with the contents of &D3. The pattern byte contains a bit set where the pixel is to be the foreground colour, and a bit clear where the pixel is to be the background colour. In four and sixteen colour modes, the pattern byte is expanded before using these locations to take account of the extra bits per pixel.

&D4 and **&D5** are similar in function to locations &D2 and &D3, only they are the graphics colour bytes. By performing an OR operation, and then an EOR operation, all the GCOL plotting operations can be taken into account by changing the data in these two bytes. The graphics mask at location &D1 is used to mask out the bits in these bytes when they are used.

&D6 and **&D7** contain the address of the top line of the current graphics character cell (eight bytes long). (See location &31A)

&D8 and **&D9** contain the address of the top scan line of the current text character.

&DA-F are used as temporary workspace.

&E0 and **&E1** are used as a pointer to the row multiplication table, high bytes first. This table is used to calculate the offsets of a character row within memory, thus as an eighty column row takes 80*8=640 bytes, for eighty column modes this points to a *640 table. This table is also used for the *320 operation needed by the 40 column modes, the results being divided by two. A *40 table is pointed to when in teletext mode. The tables consist of 32 or 24 sequential entries for the *640 and *40 tables respectively. Each entry consists of two bytes of the multiplied figure, the high byte being stored first.

&E2 is the cassette filing system status byte:

bit 0	Set if the input file is open.
bit 1	Set if the output file is open.
bit 2	Not used.
bit 3	Set if currently CATaloguing.
bit 4	Not used.
bit 5	Not used.
bit 6	Set if at end of file.
bit 7	Set if end of file warning given.

&E3 is the cassette filing system options byte, as set by the `*OPT` command. The byte is organised as two nibbles, the top four bits are used for load and save operations, and the bottom four bits are used for sequential access. The format of each nibble is:

Bits 0 and 1, the least significant bits of the nibble are used to control what happens after a tape error. When accessing the EXEC file the 'retry' and 'ignore error' options are ignored, so the EXEC is always aborted. These bits have the following meanings (note the higher bit is mentioned first:

00	Ignore errors
10	Retry after an error
01	Abort after an error

Bits 2 and 3, the most significant bits of the nibble are used to control the printing of messages during access. These bits have the following meanings (note the format given is high bit, low bit):

00	No messages
10	Short messages
11	Long messages

&E4-&E6 are used as general operating system workspace.

&E7 is the auto repeat countdown timer. This is decremented at 100Hz to zero, at which point the key is re-entered into the buffer.

&E8 and **&E9** are a pointer to the input buffer into which data is entered by OSWORD &01.

&EA is the RS423 timeout counter, which can take the following values:

=1	The cassette filing system is using 6850
=0	The RS423 system holds 6850, but has timed out.
<0	The RS423 system holds 6850, but has not yet timed out.

&EB is the 'cassette critical' flag. Bit 7 is set if the cassette filing system is called whilst doing a BGET for EXEC or a BPUT for SPOOL. It is used to ensure that no messages are printed during the access.

&EC contains the internal key number of the most recently pressed key, or zero if none is currently pressed. See the table of internal key numbers in Appendix C.

&ED contains the internal key number of the first key pressed of those still pressed, or zero if one or no keys are pressed. This is used to implement two key rollover.

&EE contains the internal key number of the character to be ignored when scanning the keyboard with OSBYTE &79. Note that Acorn have allocated this location for a RAM copy of the 1MHz bus paging register (see Chapter 23). When using the 1MHz memory, OSBYTE calls &79 and &7A should not be used, as unpredictable results can occur.

&EF contains the accumulator value for the most recent OSBYTE/OSWORD.

&F0 contains the X register value for the most recent OSBYTE/OSWORD, or the stack pointer value at the last BRK instruction.

&F1 contains the Y register value for the most recent OSBYTE/OSWORD.

&F2 and &F3 are used as a text pointer for processing operating system commands and filenames.

&F4 contains a RAM copy of the number of the currently selected paged ROM. This location should always reflect the contents of the paged ROM selection latch at location &FE30.

&F5 contains the current logical speech PHROM or ROM filing system ROM number. For the ROM filing system, if it is negative, it refers to a PHrase ROM, and if positive to a paged ROM.

&F6 and &F7 are used as an address pointer into a paged ROM or a speech PHrase ROM. These locations must be used by any paged ROM service processors for service types &0D and &0E. (see Chapter 20, paged ROMs).

&F8 and &F9 are not used by OS 1.20.

&FA and &FB are used as general operating system workspace.

&FC is used as an interrupt accumulator save register. This location is only used temporarily at the very beginning of an interrupt routine while it is setting up the stack.

&FD and &FE point to the byte after the last BRK instruction, or to the language version string after a language has been selected. See the section on the BRK vector in Chapters 16 and 20 for details of the standard layout of post-BRK data. See Chapter 20 also for details of language ROM version strings.

&FF is the escape flag. Bit 7 is set if an unserviced escape is pending. Programs that could hang up, or take a very long time, should poll this bit, and exit if it is set. The tidiest way to perform such an exit is to execute a BRK with error number &11, and the message `Escape`.

Page one, &100-&1FF

Page one is used by the 6502 stack. Locations &100 upwards are also used by some service paged ROMs to save error messages in.

Page two, &200-&2FF

Page two is the main work zone of the operating system. It contains all of the main vectors and user accessible operating system variables. Page two is laid out thus:

&200-&235 are the vectors. See Chapter 16 for more info on Vectors.

&236-&28F are the main system variables, accessed by OSBYTE calls &A6 through &FF.

&290 is the VDU vertical adjust, as set by `*TV` (OSBYTE &90).

&291 is the interlace toggle flag, as set by `*TV` (OSBYTE &90).

&292-&296 and **&297-&27B** are the two stored values of the system clock, as read by `TIME`. Two values are kept, so one can be read while the other is being updated by the interrupt routines.

&29C-&2A0 are the countdown interval timer value. This is used to cause an event after a certain time has elapsed. See Chapter 18 on events, and Chapter 15 on OSWORD, for more details of using the countdown timer.

&2A1-&2B0 form the paged ROM type table, as pointed to by value read by OSBYTEs &AA and &AB. Each byte contains the ROM type of the corresponding ROM, or zero if there is no ROM in that socket. For details of ROM types, see Chapter 20..

&2B1 and **&2B2** are the INKEY countdown timer. This is used to time out an INKEY call.

&2C5-&2C7 are used as work locations by OSWORD 1.

&2B6-&2B9 are the low bytes of the most recent analogue converter values. These are in the order channel 1.2.3 and 4.

&2BA-&2BD are the high bytes of the most recent analogue converter values.

&2BE is the analogue system flag. This contains the number of the last channel to finish conversion, or zero if no channels have finished since this value was last read. This byte is read by OSBYTE &80.

&2BF-&2C8 are the event enable flags. If zero, the event is disabled, otherwise enabled. See Chapter 18 for more details on Events.

&2C9 is the soft key expansion pointer. The next byte to be expanded in a soft key is to be found at &B01+?&2C9

&2CA is the first auto repeat count. This is the next value to go into the auto repeat counter at &E7. This location can be considered a one byte queue for the counter.

&2CB-&2CD are used as workspace for two key rollover processing.

&2CE is the sound semaphore. If it is zero it means that an envelope interrupt is being processed, so another must be ignored. If it is &FF it means that the envelope software is free.

&2CF-&2D7 are buffer busy flags. Bit 7 of these bytes is set if the matching buffer is empty. For a list of buffer numbers see OSBYTE &15 (21).

&2D8-&2E0 are the buffer start indices. They contain the offset of the next byte to be removed from each buffer. The offsets are adjusted so that the highest location in the buffer has the offset &FF for all buffers irrespective of size.

&2E1-&2E9 are the buffer end indices. They contain the offset of the last byte to be entered into each buffer. If this value is the same as the start offset, the buffer is empty. If this value is less than the start offset, it means the buffer has wrapped around to the start.

&2EA and &2EB contain the block size of currently resident block of the open cassette input file.

&2EC contains the block flag of the currently resident block of the open cassette input file. See Chapter 8 for more information on the cassette format and details of the flag byte.

&2ED contains the last character in currently resident block of the open cassette input file.

&2EE-&2FF are used as an area to build OSFILE control blocks for ***LOAD** and ***SAVE**.

Page three, &300-&3FF

Page three is used for the VDU workspace, the cassette system workspace and the keyboard buffer.

Locations &300-&37F provide the VDU workspace. In examining these locations, it should be noted that there are two forms of graphic co-ordinate, internal and external. The external graphics co-ordinate is exactly that used by the PLOT command in **BASIC**. The internal graphics co-ordinate is derived from the external by taking into account the graphics origin and scaling so that it is measured in pixels horizontally and vertically. Graphics co-ordinates are stored in four bytes, with the low byte of the X co-ordinate first.

VDU workspace is laid out thus:

&300-&307 contain the current graphics window in internal coordinates:

&300,1 Left hand column in pixels.
&302,3 Bottom row in pixels.
&304,5 Right hand column in pixels.
&306,7 Top row in pixels.

&308-&30B contain the current text window in absolute characters offset from the top left of the screen:

&308 Left hand column.
&309 Bottom row.
&30A Right hand column.
&30B Top row.

&30C-&30F contain the current graphics origin in external coordinates.

&310-&313 contain the current graphics cursor in external coordinates. This is used for calculating relative PLOTs.

&314-&317 contain the old graphics cursor in internal coordinates. This is used for the generation of triangles.

&318 contains the current text cursor X co-ordinate.

&319 contains the current text cursor Y co-ordinate.

&31A contains the line within current graphics character of the current graphics point. Because the BBC microcomputer has a non linear address space for the graphics screen, it is simpler to calculate the address of the byte at the top of the character cell that contains a point, and then calculate the row within the character. Thus the location of the byte containing the current graphics point is ?&D6 + 256*?&D7 + &31A.

&31B-&31E is used either as graphics workspace or as the first part of the VDU queue.

&31F-&323 is the VDU queue. The queue is organised so that whatever the number of characters queued, the last byte queued is always at &323.

&324-&327 contain the current graphics cursor in internal coordinates.

&328-&349 is used as general graphics co-ordinate workspace.

&34A and **&34B** contain the text cursor position as an address as sent to 6845.

&34C and **&34D** contain the text window width in bytes, i.e. the number of characters wide * the number of horizontal bytes per character * 8 for graphics modes or 1 for teletext. This is used to control the number of bytes which are soft scrolled for each line of scrolling.

&34E contains the high byte of the address of the bottom of screen memory.

&34F contains the number of bytes of memory taken up by a single character. This is 8 for 2 colour modes, 16 for 4 colour modes, 32 for 16 colour modes, and 1 for teletext mode.

&350 and &351 contain the address of the top left hand corner of the displayed screen, as is sent to the 6845.

&352 and &353 contain the number of bytes taken per character row of the screen. This is 40 for teletext mode, 320 for 8K and 10K modes and 640 for 16K and 20K modes.

&354 contains the high byte of the size of the screen memory in bytes.

&355 contains the current screen mode.

&356 contains the memory map type. The contents indicate the size of the screen memory. It has the value 0 for 20K modes, 1 for the 16K mode, 2 for 10K modes, 3 for the 8K mode, and 4 for teletext. The bottom two bits of the number in this location are sent to the addressable latch on the system VIA to control the hardware wrap around on the display.

&357-&35A contain the current colours. These are stored as the value that would be stored in a byte in screen memory to completely colour that byte to the colour required. The locations are:

&357 Foreground text colour.
&358 Background text colour.
&359 Foreground graphics colour.
&35A Background graphics colour.

&35B and **&35C** contain the graphics plot mode for the foreground and background plotting respectively. These are set by the GCOL first parameter.

&35D and **&35E** are used as a general jump vector. The vector is used for decoding VDU control codes and PLOT numbers.

&35F contains a record of the last setting of the 6845 cursor start register (even if changed through `VDU 23,0`) so that the cursor can be turned off and back on tidily using `VDU 23,1`.

&360 contains the number of logical colours in the current mode minus one.

&361 contains the number of pixels per byte minus one for the current mode, or zero if text only mode.

&362 and **&363** contain the left and right colour masks, respectively. These bytes contain a bit set in each bit position corresponding to the leftmost or rightmost pixel. For example in a two colour mode, these bytes would contain &80 and &01, and in a sixteen colour mode &AA and &55.

&364 and **&365** contain the X and Y co-ordinates of the text input cursor. The input cursor is the position from which characters are COPYed.

&366 contains the character to be used as the output cursor in teletext mode (this is normally the block character &FF).

&367 contains the font flag. This byte marks whether or not a particular font zone is being taken from ROM or RAM. If a bit is set it indicates that that zone is in RAM. See OSBYTE &14 (20) for more information on fonts.

bit 7	characters 32-63 (&20-&3F)
bit 6	characters 64-95 (&40-&5F)
bit 5	characters 96-127 (&60-&7F)
bit 4	characters 128-159 (&80-&9F)
bit 3	characters 160-191 (&A0-&BF)
bit 2	characters 192-223 (&C0-&DF)
bit 1	characters 224-255 (&E0-&FF)

&368-&36E are the font location bytes. These contain the upper bytes of the addresses of the fonts for each of the 7 zones mentioned above.

&36F-&37E form the colour palette. One byte is used for each logical colour. That byte contains the physical colour corresponding to the logical colour. The bytes are stored in numerical order of logical colour.

The area of page three from &380 to &3DF is used by the cassette filing system as working storage.

&380-&39C is used to store the header block for the BPUT file. See Chapter 8 on the cassette filing system, for details of header block layout.

&39D contains the offset of the next byte to be output into the BPUT buffer.

&39E contains the offset of the next byte to be read from the BGET buffer.

&39F-&3A6 are not used in OS 1.20.

&3A7-&3B1 contain the filename of the file being BGETed.

&3B2-&3D0 contains the block header of the most recent block read:

&3B2-&3BD	Filename terminated by zero.
&3BE-&3C1	Load address of the file.
&3C2-&3C5	Execution address of the file.
&3C6-&3C7	Block number of the block.
&3C8-&3C9	Length of the block.
&3CA	Block flag byte.
&3CB-&3CE	Four spare bytes.
&3CF-&3D0	Checksum bytes.

&3D1 contains the sequential block gap as set by `*OPT 3`.

&3D2-&3DC contain the filename of the file being searched for. Terminated by zero.

&3DD-&3DE contain the number of the next block expected for BGET.

&3DF contains a copy of the block flags of the last block read. This is used to control newlines whilst printing file information during file searches.

&3E0-&3FF are used as the keyboard input buffer.

It should be noted that although OSBYTE &A0 is officially for reading VDU variables, it may be used to read any of the values in page three.

Pages four through seven, &400-&7FF

This memory is the main workspace for the currently active language such as **BASIC**, FORTH, BCPL or VIEW.

Page eight, &800-&8FF

This page is primarily buffers, but does also contain the sound processing workspace. This page is laid out thus:

&800—&83F	Sound workspace.
&840—&84F	Sound channel 0 buffer.
&850—&85F	Sound channel 1 buffer.
&860—&86F	Sound channel 2 buffer.
&870—&87F	Sound channel 3 buffer.
&880—&8BF	Printer buffer.
&8C0—&8FF	Envelope storage area, envelopes 1-4.

Page nine, &900-&9FF

This page can be used in one of three basic ways:

1) As an extended envelope storage area:

&900—&9BF	Envelope storage area, envelopes 5-16.
&9C0—&9FF	Speech buffer.

2) As an RS423 output buffer:

&900-&9BF	RS423 output buffer.
&9C0-&9FF	Speech buffer.

3) As a cassette output buffer:

&900—&9FF	Cassette output buffer.

Uses 2) and 3) are largely compatible apart from speech, as the 6850 can only be used by either the cassette or the RS423 system at any one time, and the cassette system waits until the RS423 output has timed out before taking control of the 6850. At time out, the RS423 output buffer is usually clear.

Page ten, &A00-&AFF

This page is used for either the cassette input buffer, or for the RS423 input buffer.

Page eleven, &B00-&BFF

This page is the soft key buffer. The first seventeen bytes define the start and end locations of the sixteen soft keys. The rest of the page is allocated to the keys themselves. The start offset of soft key string n is held at location &B00+n. The address of the first character of the string is &B01+?(&B00+n). The address of the last character of the string is &B00+?(&B01+n).

Page twelve, &C00-&CFF

This page contains the font for characters 224—255. Each character requires eight sequential bytes. The first byte corresponds to the top line of the character, the second for the line below, etc.

Page thirteen, &D00-&DFF

This page is used for three functions, it contains the NMI processing routine, the expanded vector set, and the paged ROM private workspace table.

&D00—&D9E	NMI routine.
&D9F—&DEF	Expanded vector set. See vectored entry to paged ROMs for details.
&DF0—&DFF	Paged ROM workspace storage locations. There is one byte per ROM, containing the upper byte of the address of the first location of its private workspace. See paged ROMs section for more details on private workspace.

The remainder of RAM, &E00-&7FFF

Location &E00 is nominally the start of user workspace (OSHWM), but OSHWM may be raised by font explosion, or by paged ROMs taking workspace.

HIMEM is the location of the start of the screen memory, it has the value &3000 for modes 0,1, and 2; &4000 for mode 3; &5800 for modes 4 and 5; &6000 for mode 6; and &7C00 for mode 7 (teletext).

Memory is thus allocated:

OSHWM—HIMEM	User programs.
HIMEM—&7FFF	Screen memory.

The ROMs

The upper 32K of address space is allocated to the various ROMs on the BBC microcomputer.

&8000—&BFFF is allocated to the paged ROMs, including the **BASIC** and Disc Filing System ROMs.

&C000—&FFFF is allocated to the operating system ROM. Locations &FC00—&FEFF are taken out to provide space for the memory mapped peripherals.

There exists near the start of the operating system ROM some tables which are used by the operating system to assist in high resolution graphics processing. The user may be interested in the existence of these tables when disassembling the operating system. While it is possible to use these tables to speed up graphics routines, it should be noted that these tables are not Acorn supported, so a copy of the tables should be taken, rather than use them directly. These tables have not moved between operating systems 1.00 and 1.20, but there is no guarantee that these locations will not move in a future operating system version.

&C31F—&C32E is a lookup table of byte masks for four colour modes. The table is normally used for writing characters to the screen. The table is used thus:

Prepare a four bit binary number, with a bit set for each pixel in a display byte to be changed. The leftmost pixel is the highest bit. Index this table with the number generated to find the mask. If this mask were stored directly in screen memory, all the 'changed' pixels will be in colour 3, and all the 'unchanged' pixels in colour 0. By simple masking operations with the AND, ORA, and EOR instructions the mask can

be used to only change those bits required, or to set a screen byte to an appropriate mix of chosen foreground and background colours.

&C32F—&C332 is a similar lookup table to that at &C31F—&C32E, but is used for sixteen colour modes. The table is only four entries long as only two pixels can be fitted into a byte. The mask byte given in the table contains colour 15 for set bits in the index, and colour 0 for cleared bits.

&C333—&C374 contains an address table for decoding VDU codes 0 through 31. This should not be used by the user as it is even more prone to changes than the other tables.

&C375—&C3B4 this contains 32 entries of a *640 multiplication table. The high byte of each entry is held first. This table can be used to find the address of the first byte of the first character on each screen row in the 20K and 16K modes, and by dividing by 2, for 10K and 8K modes.

&C3B5—&C3E6 this contains 25 entries of a *40 multiplication table. The high byte of each entry is held first. This table can be used to find the address of the first character on each screen row in the teletext mode.

&C3E7—&C3EE contain one byte per display mode, giving the number of character rows displayed minus one.

&C3EF—&C3F6 contain one byte per display mode, giving the number of character columns displayed minus one.

&C3F7—&C3FE contain one byte per display mode, giving the value stored in the video ULA control register for that mode.

&C3FF—&C406 contain one byte per display mode, giving the number of bytes storage taken per character. This is 1 for teletext, 8 for two colour modes, 16 for 4 colour modes, and 32 for 16 colour modes.

&C407—&C408 contain the mask table for sixteen colour modes. There are two entries, one per pixel in the byte. Each mask contains the value that would be stored in screen memory if the appropriate pixel were set to colour 15, and the other to zero. The left pixel is stored in the first byte of the table, and the right in the second.

&C409—&C40C contain the mask table for four colour modes. There are four entries, one per pixel in the byte. Each mask contains the value that would be stored in screen memory if the appropriate pixel were set to colour 3, and all the others to zero. The leftmost pixel is stored first byte in the table, and the rightmost in the last.

&C40D—&C414 contain the mask table for two colour modes. There are eight entries, one per pixel in the byte. Each mask contains the value that would be stored in screen memory if the appropriate pixel were set to colour 1, and all the others to zero. The leftmost pixel is stored first byte in the table, and the rightmost in the last.

&C414—&C41B Note that this table overlaps the last. This table contains one byte per mode containing the number of colours available minus one.

&C41B—&C425 Note that this table overlaps the last. This table contains four five byte tables used to process the five GCOL plotting options. These tables are highly overlapped.

&C424—&C439 are the colour tables. There is a colour table for each of: 2 colours, 4 colours and 16 colours. Each table contains one byte per colour, in ascending order of colour number. The byte contains the value that would be stored in screen memory to give a fully coloured byte of that colour. These colour bytes are used in conjunction with the various masks mentioned above.

&C424—&C425 The two colour table.

&C426—&C429 The four colour table.

&C42A—&C439 The sixteen colour table.

&C43A—&C441 contain one byte per display mode, giving the number of pixels per byte on the screen minus one. The value stored is zero for non graphics modes.

&C440—&C447 contain one byte per display mode, giving the memory map type for the mode. This is zero for 20K modes, 1 for the 16K mode, 2 for 10K modes, 3 for the 8K mode, and 4 for teletext. Note that this table overlaps the last.

&C447—&C458 contain various VDU section control numbers, of no direct relation to any screen parameters. Note that this zone overlaps the previous table.

&C459—&C45D contain one byte per memory map type (see above) giving the most significant byte of the number of bytes taken up by the screen. The least significant byte is always zero.

&C45E—&C462 contain one byte per memory map type (see above) giving the most significant byte of the address of the first location used by the screen. The least significant byte is always zero.

&C463—&C46D contain tables used by the VDU section to index into the other tables.

&C46E—&C4A9 contain tables of values to be sent to the various 6845 registers. There is one block of 12 bytes for each of the five memory map types (see above).

&C46E—&C479 6845 registers 0-11 for memory map type 0 (modes 0-2).

&C47A—&C485 6845 registers 0-11 for memory map type 1 (mode 3).

&C486—&C491 6845 registers 0-11 for memory map type 2 (modes 4-5).

&C492—&C49D 6845 registers 0-11 for memory map type 3 (mode 6).

&C49E—&C4A9 6845 registers 0-11 for memory map type 4 (mode 7).

&C4AA—&C4B5 are used by the VDU driver as Jump table addresses into its internal plotting routines. They should not be used by the user at all.

&C4B6—&C4B9 are a teletext conversion table. The Teletext character generator uses slightly different character codes to the rest of the BBC microcomputer. This table contains four bytes, an ASCII value followed by the byte to be stored in screen memory, for three characters.

&23 (#) is translated to &5F
&5F (_) is translated to &60
&60 (£) is translated to &23

18. Events and Event handling

The concept of events and event handling provides the user with an easy to use, pre-packaged interrupt. A routine may be written to perform a second function while another program is running. Because the microprocessor can only do one thing at a time the second function will be executed by interrupting the first program and then returning control to allow it to continue from where it was interrupted.

The interrupt facility is provided by the designers of the microprocessor and is a fairly primitive level of control. The operating system uses interrupts extensively and also provides the user with the ability to trap interrupts (see Interrupts, Chapter 19).

The operating system interrupts whatever is going on in the foreground (e.g. the user program) every 10 milliseconds and performs any tasks which are required. This background work includes controlling the sound chip, storing key strokes in the input buffer and keeping up with ADC conversions. In the process of performing the background processing, the operating system may generate a number of events which hand the processor over to an event handling routine provided by the user. Thus the user is able to append some code of his own to the operating system's interrupt handling routine.

The events available are:

Event number	Cause of event
0	Output buffer becomes empty
1	Input buffer becomes full
2	Character entering input buffer
3	ADC conversion complete
4	Start of vertical sync
5	Interval timer crossing zero
6	ESCAPE condition detected
7	RS423 error detected.
8	Econet generated event.
9	User event.

OSBYTE calls &D/*FX 13 and &E/*FX 14

Disable/enable event

These calls are used to switch particular events on and off. On entry both these calls require the event to be specified by its event number in X. OSBYTE &D/`*FX 13` can be used to disable a particular event and OSBYTE &E/`*FX 14` can be used to enable each event. Even though events continue to be generated when disabled they will not be passed on to the event handling routine.

The Event Vector (EVNTV)

Address &220

When any event is enabled the operating system jumps (using JSR) to the address contained in EVNTV. To interface the user's event handling routine the entry address of the routine must be placed in the EVNTV.

Event handling routines

The user's event handling routine is entered with the accumulator containing the event number. Other information may also be passed to the event handling routine in X or Y; this is specific for each event (see below).

The event handling routine should preserve all registers.

The event handling routine is entered with interrupts disabled and should not enable interrupts. Because interrupts are disabled, an overly long routine will have dire consequences and the routine should be terminated after no more than about 2 milliseconds.

Great care must be taken when using operating system calls from within an event handling routine. Many operating system calls may enable interrupts during execution. If an interrupt occurs while the user's event handling routine is being executed the interrupt may cause the user's routine to be re-entered before it has finished processing the previous event. It may be possible to write the event handling routine in such a way that this will not have any ill effects. A routine which may be re-entered in this way is described as re-entrant or re-enterable. While the event facility has been designed to enable users easy access to a form of interrupt handling this aspect of their use may complicate the issue. For more information see Interrupts, Chapter 19.

The event handling routine should never be exited using an RTI instruction. Using the old contents of the event vector is a good way of leaving the routine. Making a copy of the vector contents before changing this vector allows the user routine to JMP indirect when the new routine has finished. Using this method allows more than one event handling routine to be used at one time. An RTS instruction may be used to exit the routine if no other event handling is to be allowed.

Event Descriptions

0 - Output buffer empty

This event enters the event handling routine with the buffer number (see OSBYTE &15/*FX21) in X. It is generated when a buffer becomes empty (i.e. just after the last character is removed).

1 - Input buffer full

This event enters the event handling routine with the buffer number (see OSBYTE &15/*FX21) in X. It is generated when the operating system fails to enter a character into a buffer because it is full. Y contains the character value which could not be inserted.

2 - Character entering input buffer

This event is normally generated by a key press and the ASCII value of the key is placed in Y. It is generated independently of the input stream selected.

For example:

```
 10 OSWORD=&FFF1
 20 EVNTV=&220
 30 DIM MC% 100
 40 DIM sound_pars 8
 50 FOR I=0 TO 3 STEP3
 60     P%=MC%
 70     [
 80     OPT I
 90     PHP
100     PHA
110     TXA
120     PHA
130     TYA
140     PHA                     \ save registers
150     STY sound_pars+4        \ SOUND pitch=key ASCII value
160     LDX #sound_pars AND 255
170     LDY #sound_pars DIV 256
180     LDA #7
190     JSR OSWORD              \ perform SOUND command
200     PLA
210     TAY
220     PLA
230     TAX
240     PLA
```

```
250     PLP                     \ restore registers
260     RTS                     \ return from event handler
270 ]
280 NEXT I
290 ?EVNTV=MC% AND &FF
300 EVNTV?1=MC% DIV &100
310 !sound_pars=&FFF50001
320 sound_pars!4=&00010000 :REM set up SOUND 1,-11,x,1
330 *FX 14,2
340 :REM enable keyboard event
```

This example program illustrates how the keyboard event can be used. When this program has been run, each key press causes a sound to be made. The pitch of this sound is dependent on the key pressed. This event handling routine does not check the identity of the event calling it and so enabling events other than the keyboard event will have curious consequences.

3 - ADC conversion complete

When an ADC conversion is completed on a channel this event is generated. The event handling routine is entered with the channel number on which the conversion was made in Y.

4 - Start of vertical sync

This event is generated 50 times per second coincident with vertical sync. One use of this event is to time the change to a 6845 or video ULA register so that the change to the screen occurs during fly back and not while the screen is being refreshed. This avoids flickering on the screen.

5 - Interval timer crossing zero

This event uses the interval timer (see OSWORD calls &3 and &4). This timer is a 5 byte value incremented 100 times per second. The event is generated when the timer reaches zero.

For example:

```
 10 MODE7
 20 OSWORD=&FFF1
 30 OSBYTE=&FFF4
 40 OSWRCH=&FFEE
 50 EVNTV=&220
 60 DIM MC% 100
 70 DIM clock_pars 5
 80 DIM count 1
 90 FOR I=0 TO 2 STEP 2
100     P%=MC%
110     [
120     .entry OPT I
130     PHP
140     PHA
150     TXA
160     PHA
130     TYA
180     PHA                     \ save registers
190     LDX #clock_pars AND 255
200     LDY #clock_pars DIV 256
210     LDA #4
220     JSR OSWORD              \ write to interval timer
230     LDA #&86
240     JSR OSBYTE              \ read text cursor position
250     TYA                     \ and save it on the stack
260     PHA
270     TXA
280     PHA
290     LDA #31                 \ reposition text cursor
300     JSR OSWRCH              \ with VDU 31,38,1
310     LDA #38
320     JSR OSWRCH
330     LDA #1
340     JSR OSWRCH
350     LDA count               \ put count in A
360     JSR OSWRCH              \ write it not
370     CMP #ASC"9"     \ has count reached 9
380     BNE over                \ jump next bit if it hasn't
390     LDA #ASC"/"
400     STA count               \ put '-1' in count
410     .over INC count         \ count=count+1
420     LDA #31                 \ restore old cursor position
430     JSR OSWRCH              \ using VDU 31
440     PLA
450     JSR OSWRCH
460     PLA
470     JSR OSWRCH
```

```
480     PLA
490     TAY
500     PLA
510     TAX
520     PLA
530     PLP                     \ restore registers
540     RTS                     \ return from event handler
550     ]
560 NEXT I
570 ?EVNTV=MC% AND &FF
580 EVNTV?1=MC% DIV &100
590 !clockpars=&FFFFEF9C
600 clock_pars?4=&FF : REM clock value -100 centiseconds
610 *FX 14,5
620 REM interval timer event
630 ?count=ASC"0" : REM initialise count
640 CALL entry : REM initialise clock
```

This demonstration program uses the interval timer event to call the event handling routine at one second intervals. The 5 byte clock value is incremented every centisecond and so the first task the routine must perform is to write a new value (-100) to the timer to prepare for the next call. Each time the routine is entered a count is incremented and the ASCII character printed up on the top right corner of the screen. In this simple example repeating a count from &30 to &39 (ASCII '0' to '9') makes the programming easy. It does not require a lot of imagination to see how this concept could be expanded to provide a constant digital time read out.

6 - ESCAPE condition detected

When the **<ESCAPE>** key is pressed or an ESCAPE is received from the RS423 (when RS423 ESCAPEs are enabled) this event is generated. When this event is enabled, the ESCAPE state (indicated by the flag at &FF) is not set when an ESCAPE condition occurs. Therefore after a `*FX136` the **<ESCAPE>** key has no effect in **BASIC**.

7 - RS423 error

This event is generated when an RS423 error is detected. This event is entered with the 6850 status byte shifted right by one bit in the X register and the character received in Y.

8 - Network error

This event is generated when a network event is detected. If the net expansion is not present then this could be used for user events.

9 - User event

This event number has been set aside for the user event. This is most usefully generated from a user interrupt handling routine to enable other user software to trap an interrupt easily (e.g. an event generated from an interrupt driven utility in paged ROM). An event may be generated using OSEVEN.

19. Interrupts

An interrupt is a hardware signal to the microprocessor. It informs the 6502 that a hardware device, somewhere in the system, requires immediate attention. When the microprocessor receives an interrupt, it suspends whatever it was doing, and executes an interrupt servicing routine. Upon completion of the servicing routine, the 6502 returns to whatever it was doing before the interrupt occurred.

A simple analogy of an interrupt is a man working hard at his desk writing a letter (a foreground task). Suddenly the telephone rings (an interruption). The man has to stop writing and answer the telephone (the interrupt service routine). After completion of the call, he has to put the telephone down, and pick up his writing exactly where he left off (return from interrupt).

In a computer system, the main objective is to perform foreground tasks such as running **BASIC** programs. This is equivalent to writing the letter in the above example. The computer may however be concerned with performing lots of other functions in the background (equivalent to the man answering the telephone). A computer which is running the house heating system for example would not wish to keep on checking that the temperature in every room is correct - it would take up too much of its processing time. However, if the temperature gets too high or too low in any of the rooms it must do something about it very quickly. This is where interrupts come in. The thermostat could generate an interrupt. The computer quickly jumps to the interrupt service routine, switches a heater on or off, and returns to the main program.

There are two **BASIC** types of interrupts available on the 6502. These are maskable interrupts (IRQs) and non maskable interrupts (NMIs). To distinguish between the two types, there are two separate pins on a 6502. One of these is used to generate IRQs (maskable) and the other is used to generate NMIs (non maskable).

Non Maskable Interrupts

When a non maskable interrupt is asserted by a hardware device connected to the NMI input on the 6502, a call is immediately made to the NMI service routine at &0D00 on the BBC microcomputer. Nothing in software can prevent this from

happening. So that the 6502 is only interrupted in very urgent situations, only very high priority devices such as the Floppy Disc Controller chip or Econet chip are allowed to generate NMIs. They are then guaranteed to get immediate attention from the 6502. To return to the main program from an NMI, an RTI instruction is executed. It is always necessary to ensure that all of the 6502 registers are restored to their original state before returning to the main program. If they are modified, the main program will suddenly find garbage in its registers in the middle of some important processing. It is probable that a total system 'crash' would result from this.

Maskable Interrupts

Maskable interrupts are very similar to non-maskable interrupts in most respects. A hardware device can generate a maskable interrupt to which the 6502 must normally respond. The difference comes from the fact that the 6502 can choose to ignore all maskable interrupts under software control if it so desires. To disable interrupts (only the maskable ones though), an SEI (set interrupt disable flag) instruction is executed. Interrupts can be re-enabled at a later time using the CLI (clear interrupt disable flag) instruction.

When an interrupt is generated, the processor knows that an interrupt has occurred somewhere in the system. Initially, it doesn't know where the interrupt has come from. If there were only one device that could have caused the interrupt, then there would be no problem. However, since there is more than one device causing interrupts in the BBC microcomputer, each device must be interrogated. That is, that each device is asked whether it caused the interrupt.

When the interrupt processing routine has discovered the source of a maskable interrupt, it must decide what type of action is required. This usually involves transferring some data to or from the interrupting unit, and clearing the interrupt condition. The interrupt condition must be cleared because most devices that use interrupts continue to signal an interrupt until they have been serviced. The completion of servicing often has to be signalled by the processor writing to a special register in the device.

Interrupts must not have any effect on the interrupted program. The interrupted program will expect the processor registers and flags to be exactly the same after

return from an interrupt routine as they were before the interrupt occurred. Thus an interrupt routine must either not alter any registers (which is difficult) or restore all register contents to their original values before returning.

Interrupt routines are entered with interrupts disabled, so a second interrupt cannot occur whilst an interrupt routine is still processing, unless interrupts are deliberately enabled because the interrupt servicing is likely to take an appreciable time. When this is done, the interrupt routine must be written with care because the interrupt service routine can then itself be interrupted. For this reason it is usual to save register contents onto the processor stack, rather than to fixed memory locations which may get overwritten in a subsequent incarnation of the interrupt routine.

Interrupts on the BBC microcomputer

Interrupts are required on the BBC microcomputer to process all of the 'background' operating system tasks. These tasks include incrementing the clock, processing envelopes or transferring keys pressed to the input buffer. All of these tasks must continue whilst the user is typing in, or running his program. The use of interrupts can give the impression that there is more than one processor, one for the user, one updating the clock, one processing envelopes, etc.

As was mentioned in the introduction, normal (maskable) interrupts may be disabled. Care should be taken to ensure that interrupts are not disabled for a long time. If they are then the operating system will cease to function properly. Interrupts should only be disabled for such critical things as changing the two bytes of a vector, writing to the system VIA (see the system VIA in Chapter 23) or handling an interrupt or event. Interrupts should not be disabled for long, because whilst interrupts are disabled, the clock stops, and all other interrupt activity ceases. Interrupts are disabled by the SEI assembler instruction, and re-enabled with CLI. Most devices that generate interrupts will continue to signal an interrupt until it is serviced, and so will wait through the period of interrupts being disabled. For this reason most short periods of interrupts disabled are safe, it is only if a second interrupt occurs from a device before the first is serviced that problems can occur.

Using Non—Maskable Interrupts

Generally, NMIs are reserved for specialised pieces of hardware which require very fast response from the 6502. NMIs are not used on a standard system. They are used in floppy disc and Econet systems (but not on hard disk or CF systems). An NMI causes a jump to location &0D00 to be made.

Using Maskable Interrupts

Most of the interrupts on the BBC microcomputer are maskable. This means that a machine-code program can choose to ignore interrupts if it wishes, by disabling them. Since all of the operating system features such as scanning the keyboard, updating the clock and running the serial system are run on an interrupt basis, it is unwise to disable interrupts for more than about 2ms.

There are two levels of priority for maskable interrupts, defined by two indirection vectors in page &02. The priority of an interrupt indicates its relative importance with respect to other interrupts. If two devices signal an interrupt simultaneously, the higher priority interrupt is serviced first.

Interrupt Request Vector 1 (IRQ1V)

This is the highest priority vector through which all maskable interrupts are indirected. This is nominally reserved for the system interrupt processing routine. This operating system routine handles all anticipated internal IRQs. Anticipated IRQs include interrupts from the keyboard, system VIA, serial system and the analogue to digital converter. Any interrupt which cannot be dealt with by the operating system routine (such as an interrupt from the user VIA or a piece of specialised hardware) is passed on through the second interrupt vector.

Within this interrupt routine the devices are serviced in the following order of priority, highest first:

1) The 6850 serial chip
2) The system 6522 VIA
3) The user 6522 VIA

Interrupt Request Vector 2 (IRQ2V)

Any interrupts which cannot be dealt with by the operating system are passed on through this lower priority vector. This vector is reserved for user supplied

interrupt routines. The user should intercept the interrupts at this point, rather than at IRQ1V whenever possible. It should only be necessary to intercept IRQ1V when a very high priority is required by the user routine.

Note that the user supplied routine must return control to the operating system routine to ensure clean handling of interrupts. It is therefore advisable to store the original contents of the indirection vector in memory somewhere. This will enable the user routine to jump to the correct operating system routine. Also, by using this method of jumping to the old contents of the vector, several user routines can all intercept it correctly.

Operating system interrupt processing

The following sections describe how the operating system deals with interrupts indirected via IRQ1V. In very specialised circumstances, the programmer may wish to process these interrupts himself in some special way.

Serial interrupt processing

The 6850 asynchronous communications interface will produce three types of interrupt:

1. Receiver interrupt - a character has been received.
2. Transmitter interrupt - a character has been transmitted.
3. Data Carrier Detect (DCD) interrupt - a 2400Hz tone has been discontinued - at the end of a cassette block.

The 6850 contains a status byte that enables the 6502 to locate the cause of the interrupt. This byte is organised as:

bit 0	This bit is set on a receiver interrupt. Bits four five and six are valid after this interrupt.
bit 1	This bit is set on a transmit interrupt.
bit 2	This bit is set on a DCD (data carrier detect) interrupt.
bit 3	This bit is set if the 6850 is not CLEAR TO SEND.
bit 4	Framing error. Receive error.
bit 5	Receiver overrun. Receive error.
bit 6	Parity error. Receive error.

bit 7 Set if the 6850 was the source of the current interrupt. This bit is the first to be checked by the operating system interrupt handling routine. If it isn't set, the routine moves on to checking the system VIA to see if it generated the interrupt.

Serial processing can be split into two parts, that done for the cassette filing system, and that done for the RS423 system.

The cassette serial system

The cassette system uses the interrupts in the following ways:

- A transmitter interrupt causes the next byte of output data to be sent to the 6850.
- A receiver interrupt causes a byte to be taken from the 6850 and stored in memory.
- A Data Carrier Detect interrupt is used to mark the end of a data block when skipping to find files or during a cataloging process.

The RS423 serial system

The RS423 system uses the interrupts in the following ways:

A transmitter interrupt causes a character to be sent to the 6850 from the RS423 transmit buffer, or the printer buffer if the RS423 printer is selected. If both buffers are empty, the RS423 system is flagged as available (see OSBYTE &BF) and transmitter interrupts are disabled.

A receiver interrupt is used to cause a character to be read from the 6850 and inserted into the RS423 receive buffer (if enabled by use of OSBYTE &9C). If there is a receive error, (which can be ignored by use of OSBYTE &E8) event number 7 is generated, and the character is ignored. The character is also ignored if OSBYTE &CC has been made non-zero. The RTS line is pulled high if the receive buffer is getting full. The number of characters which need to be in the buffer to cause RTS to go high is set by OSBYTE &CB.

A DCD interrupt cannot occur unless the RS423 has been switched to the cassette connector by use of OSBYTE &CD. The DCD interrupt is normally cleared by reading from the 6850 receive register. An event number 7 (RS423 receive error event) is then generated.

The RS423 system can be made to ignore any of the above interrupts by use of OSBYTE &E8. The 6850 status register is ANDed with the OSBYTE value. Any bit cleared by this is ignored, and passed over to the user interrupt vector. The user is then responsible for clearing the interrupt condition. This is done by either reading the receive data register or writing to the transmit data register of the 6850.

System VIA interrupt processing

The system VIA controls and monitors many of the BBC microcomputer's internal hardware devices. An interrupt generated by the system VIA may have many different interpretations. The reader is referred to VIAs in general, and the system VIA in particular, in Chapter 23.

When it generates an interrupt, the system VIA's status byte indicates the type of interrupt:

bit 0	Set if a key has been pressed.
bit 1	Set if vertical synchronisation has occurred on the video system (a 50Hz time signal).
bit 2	Set if the system VIA shift register times out. This should not normally occur since the shift register is not used on the system VIA.
bit 3	Set if a lightpen strobe off the screen has occurred.
bit 4	Set if the analogue converter has finished a conversion.
bit 5	Set if timer 2 has timed out. Used for the speech system.
bit 6	Set if timer 1 has timed out. This timer provides the 100Hz signal for running the internal clocks.
bit 7	Set if the system VIA was the source of the interrupt.

The standard interrupt routine can be made to ignore any of the above interrupts by use of OSBYTE &E9. The status register is ANDed with this OSBYTE value. Any bit masked by this is ignored, and passed over to the user interrupt vector. The user is then responsible for clearing the interrupt condition. This is done by writing a byte to location &FE4D with the bit corresponding to the interrupt to be cancelled set.

The standard interrupt routine uses the interrupts in the following ways:

A key pressed interrupt causes the operating system to mark the key pressed as the current key. It will be processed on a subsequent timer interrupt.

A vertical sync interrupt is used to time the colour flashing and change the colours when required. Modifying colours in this way ensures that the colours do not change halfway through displaying the screen. It is also used to 'time out' the cassette and RS423 systems (when one of these has timed out after half a second of inactivity, the 6850 can be claimed for use by the cassette or RS423 systems). This interrupt also causes a frame sync event (number 4).

The shift register of the system VIA is not used, and its interrupt is passed over to the user.

The analogue conversion completion interrupt is used to cause the newly converted value to be read, and stored in RAM. The next channel to be converted is then initialised. This interrupt also causes event number 3.

The light pen interrupt is not used by the operating system, as it has no software to support the light pen. This interrupt is always passed over to the user.

Timer 2 is used by the operating system to count transitions of the speech 'ready' output. When an interrupt occurs, there is an attempt to speak another word.

Timer 1 is used to provide regular 100Hz interrupts. On receipt of this interrupt, the following happens:

a) The interrupt is cleared.
b) The 'TIME' clock is swapped with its alternate and incremented. There are two 5 byte clocks provided by the operating system. They are updated alternately. This ensures that any program which is in the middle of reading the clock when an interrupt occurs can still read a valid value.
c) The interval timer is incremented, and if zero an event is caused (number 5).
d) The INKEY timer is decremented.
e) One element of sound processing is performed.
f) If a new key has been pressed, its code is entered into the buffer, and auto repeat processing is begun.
g) Then three emergency back-door operations are performed:

a. The speech chip is checked, in case a speech interrupt has been missed.
b. The 6850 is checked, because the transmitter interrupts must be disabled to set the RTS line high.
c. The analogue converter is checked, to ensure that an interrupt is not missed.

If system VIA interrupts are disabled, sound, speech, the clock, the countdown timer, INKEY, the keyboard, colour flash, and analogue converters will all cease to work. It is therefore inadvisable to disable interrupts for more than about 2ms at any time.

User VIA interrupt processing

Port B of the user VIA is reserved for user applications, and port A is used as a parallel printer interface.

The standard interrupt routine only uses the CA1 interrupt; all others are passed over to the user. The CA1 interrupt is used to signify that the parallel printer is ready to accept a new character. A new character is sent to the printer if the printer output buffer is not empty. OSBYTE call &E7 can be used to mask out the CA1 interrupt and cause it to be passed on to the user in the same way as interrupts are masked out of the system VIA.

Intercepting interrupts

If the user intercepts either IRQ1V or IRQ2V by changing the vector value at &204,5 or &206,7, the following conditions apply on entry to the user's interrupt routine:

- The original processor status byte and return address are already stacked ready for an RTI instruction.
- The original X and Y states are still in their registers.
- The original A register contents are in location &FC.

Note that the interrupt routine should not call any operating system routines if at all possible. This is because it is possible that the foreground process was using the

desired routine at the time of the interrupt. If the routine to be called is not re-entrant, the foreground process will be disturbed, and may crash.

The user's interrupt routine should be 're-entrant'. This means that if any routines called by the interrupt routine re-enable interrupts, and a second interrupt occurs before the first is finished, the interrupt routine should be able to handle it. This is achieved by:

- Pushing the original X, and Y registers onto the stack.
- Pushing the contents of &FC onto the stack.
- Not using any absolute temporary storage locations.

Note that enabling interrupts during a user interrupt routine is unofficial, in that Acorn state that it should not be done, but with care it is safe to do so.

If a non-re-entrant zone is entered (such as an operating system routine, or an area that needs temporary storage), keep a semaphore for that zone. If another interrupt then occurs, that area of code must not be used again until it has finished.

20. Paged ROMs

Paged ROMs are ROMs that fit in the four rightmost ROM sockets on the BBC microcomputer circuit board. They all have the following features in common:

They exist in the address space &8000 to &BFFF; thus only one can be active at a time, the rest being 'paged' out.

They are scanned by the operating system under certain circumstances, usually associated with an occurrence which is not understood by the operating system, e.g. inexplicable interrupts, unrecognised operating system commands.

The operating system has the software to handle up to 16 paged ROMs, although unmodified, the hardware can only handle four.

A paged ROM is used in the following applications:

- Filing systems, such as disc operating systems, or the Econet system.
- Languages, such as **BASIC**, FORTH, BCPL, etc.
- Utilities, such as wordprocessors, debugging aids, file utilities etc.
- Hardware drivers, for personalised hardware on the 1MHz bus, such as the RetroClinic USB DataCentre, or the BeebSID music generator.

A paged ROM must be recognised by the operating system. To be recognised the first few bytes must conform to:

00-02	JMP language entry
03-05	JMP service entry
06	ROM type
07	Copyright offset pointer (=nn)
08	Binary version number
09..	Title string, printed on selection as a language.
vv..	Optional version string, preceded by &00.
nn-nn+3	&00, &28 '(', &43 'C', &29 ')'
nn+4..	Copyright message
xx	Copyright message terminator (&00)
xx+1-xx+5	If applicable, second processor relocation address.

a) The language entry is called upon initialising the ROM as a language, and after copying over to the second processor, if applicable.
b) The service entry is called regularly, whenever a service is needed by the MOS.
c) The ROM type is a flag byte informing the MOS as to what the ROM is expected to do.

bit 7 — If set, this indicates that the ROM has a service entry. Note that a ROM with no service entry is assumed to be the **BASIC** ROM. ALL user ROMs should have a service entry.

bit 6 — If set, this indicates that the ROM has a language entry, and wishes to be considered for selection on a hard reset. If not set, the ROM may still have a language entry, but it must be started up by an operating system command; such a language, if selected, will still be selected after a soft reset.

bit 5 — If set, this indicates that a second processor relocation address is provided, and that the code in this ROM, excepting that for the service entry, has been assembled from that address. This is only relevant if there is a language resident in the ROM; service routines are never copied across the Tube.

bit 4 — This bit controls the Electron soft key expansions, and is not relevant on the BBC microcomputer.

bit 1 — Must be set.

d) The copyright offset pointer is the offset from the beginning of the ROM to the zero byte preceding the copyright message.
e) The binary version number is ignored by the operating system. It should be used to indicate the version number to anyone examining the ROM.
f) The title string is printed out on selection of the ROM as a language ROM. Apart from this, its only use is to identify the ROM to those examining the ROM.
g) The version string is optional, and if it exists, it must be preceded by a zero byte. It is only really of relevance to languages, because on entry to a language the error pointer (&FD and &FE) will point to this, or if not

present, the copyright message. `REPORT` in **BASIC** demonstrates that **BASIC** has no version string.

h) The copyright string is essential. This is because the ROM is recognised by this string, and if it does not exist, the ROM will be ignored. The format must always be a zero byte followed by '`(C)`'.

i) The tube relocation address, if present, indicates to the tube system that the language part of the contents of this ROM have been assembled to an address other than &8000. The address given is the address to which the program must be copied in the second processor. The service call code should still be assembled into the &8000 space, because, for service entries, the ROM is executed within the I/O processor.

Entering Paged ROMs

Paged ROMs are entered via one of three methods: by a service call, via an extended vector, or through the language entry point. Generally, a ROM should never be executed at all until it has responded to at least one service call; the exception to this is **BASIC** which has no service entry point. Always when within a paged ROM, location &F4 contains the ROM number of the ROM being executed.

Service Call entries

When the paged ROMs are asked to provide a service, the highest priority ROMs (the ones in the sockets to the right of the board) are offered the chance first. They are entered at the service entry point with the registers set up thus:

A	Service type requested
X	ROM number of the current ROM
Y	Any parameter required for the service

If a ROM wishes to issue further service calls to other ROMs, for example to claim memory, it should issue such calls using OSBYTE call &8F, with the service type in the X register and the parameter in the Y register.

If a ROM does not wish to provide the service, it should exit with all the registers preserved. If, however, the ROM performs the requested service, and wishes to prevent other ROMs also performing the service, the accumulator should be zero on exit.

The service types are:

00 No operation. All ROMs are to ignore this service request: a higher priority ROM has already provided it.

01 Absolute workspace claim. On break, each ROM is asked to stake a claim for absolute workspace. Absolute workspace is a single block of memory which is only allocated to one ROM at any one time. Being absolute, this memory runs from &E00 to the highest address asked for by any of the ROMs. On entry, the Y register contains the current upper limit of the absolute workspace. This starts off as &0E (the upper byte of the first address of absolute workspace). Each ROM should compare the value in the Y register with the upper limit of absolute memory required by the ROM, and if necessary replace it by the level required by the ROM. This memory is shared between all the ROMs, and should be claimed with service call &0A before use. The accumulator should be preserved during this call.

02 Private workspace claim. On break, after absolute workspace allocation, each ROM is offered the chance to take some private workspace. This memory is exclusive to the ROM claiming it. The ROM is entered with the first page number of the workspace available to it in the Y register. It should save this value in the ROM workspace table at &DF0 to &DFF (for ROMs 0 to &F respectively), and add to the Y register the length in 256 byte pages of the private workspace required. Because the absolute workspace is shared between all the paged ROMs, each ROM that uses it should keep a flag within its private workspace which indicates whether or not it currently has control of the absolute workspace. This enables the ROM to claim the workspace when it needs it, and to respond to such a claim from other ROMs.

03 Auto-boot. Each service ROM is given the opportunity to initialise itself on break. This is primarily used for filing systems to set up their vectors on break rather than having to be reselected every time with an operating system command. To allow lower priority ROMs a look in, each ROM should examine the keyboard before initialisation, and initialise only if no key is pressed, or a key exclusive to that ROM is pressed (eg. the discs are selected by 'D-break'). If the ROM initialises, it should look for, and RUN,

EXEC or LOAD, a boot file (typically called '!BOOT') if on entry the Y register is zero.

04 Unrecognised command. When the user issues an OS command that is not recognised by the central operating system, the command is first offered to the paged ROMs, and then to the currently active filing system. ROMs containing general utilities should use this call to activate them. Filing system and language ROMs should trap this to catch their selection command. Note that most filing system commands should be intercepted via the filing system control entry. On entry, the command to be interpreted is in the form of an ASCII string terminated by &0D and pointed to by the contents of &F2 and &F3 plus the Y register.

05 Unrecognised interrupt. When an interrupt occurs that is either not recognised by the operating system, or has been software masked out, the interrupt is first offered to the paged ROMs, and then to the user via the 'IRQ2' vector. A paged ROM accepting interrupts should interrogate the device(s) that it will respond to, to see if any of them are responsible for the interrupt, and if so process it. If the interrupt is processed by a ROM, it should set the accumulator to zero to prevent it being offered elsewhere. Always return with an RTS instruction, not RTI.

06 Break. The user has executed a BRK instruction, usually flagging an error. Paged ROMs are informed before handing over the error to the current language via the break vector. The Y register should be preserved during this call, but only if the service routine intends to return and allow control to pass back to the current language. On entry location &F0 contains the value of the stack pointer after the BRK instruction was executed. Locations &FD and &FE point to the error number in memory. Note that the error may have occurred in another ROM, whose contents are not directly accessible by the service routine. OSBYTE &BA will give the ROM number of the ROM which was active when the BRK occurred.

07 Unrecognised OSBYTE call. The user has issued an OSBYTE call not known to the operating system. The A, X and Y registers on entry to the OSBYTE are stored at &EF, &F0 and &F1 respectively. The OSBYTE call should be recognised on the contents of &EF.

08 Unrecognised OSWORD call. The user has issued an OSWORD call not known to the operating system. The A, X and Y registers on entry to OSWORD are stored at &EF, &F0 and &F1 respectively. The OSWORD call should be recognised on the contents of &EF. Note that it is not worth trapping OSWORD calls with numbers greater than &E0, as these are all sent to the user vector at &200. Note also that OSWORD number 7 (make a sound) will cause this service call if an unrecognised channel number is used (in the range &2000 to &FFFF), allowing for future sound channel expansion over the 1MHz bus.

09 ***HELP** instruction expansion. This service call is made whenever the user issues a ***HELP** command. ROMs should allow all resident ROMs to respond to it at once, so that the user can find out about all the resident ROMs at once. On entry, the rest of the command after the ***HELP** is pointed to by the contents of locations &F2 and &F3 plus the Y register. ROMs should recognise keywords on that line to provide information on a particular area. If the rest of the line is blank, the ROM should type its name, and a list of keywords to which it will respond.

0A Claim static workspace. When a paged ROM requires use of the absolute workspace starting at location &E00, it should ask the current owner to relinquish it by issuing this call. On receiving this call, a ROM should copy its valuable information to its private area, and update its flag in its private area to indicate that it no longer owns the workspace. When it again needs the absolute workspace, it should itself issue this call to get it back.

0B NMI release. This call, when issued, means that the current user of the NMI space no longer requires it, and it may be claimed. The Y register on entry should contain the ROM number of the previous owner (usually the net system), and each ROM should compare it with their own ROM number (in the X register and &F4), before reasserting control over the NMI space.

0C NMI claim. This call should be made with Y=&FF, and if a ROM is currently the owner of the NMI space, it should return in the Y register its ROM number, and clear from the NMI space any important data. Y should not be altered if the NMI space was not previously in use. The claimer of NMI

should store the returned ROM number for use when releasing the NMI claim.

0D ROM filing system initialise. This call is issued when the ROM filing system is active, and the paged ROMs are being scanned for a particular file. On entry, the Y register contains 15 minus the ROM number of the next ROM to be scanned. If that adjusted ROM number is less than the number of the ROM receiving the call, the call should be ignored. Otherwise, the number should be replaced by the ROM number of the active ROM and stored at &F5 after adjusting it. The current ROM should mark itself as active by returning with the accumulator zero, and store in locations &F6 and &F7 a pointer to the data within the ROM. The alteration of location &F5 is necessary because the ROM filing system will abort a search if it gets no response from any of the ROMs for any particular ROM number. Altering &F5 causes non-existent ROMs to be skipped over.

0E ROM filing system byte get. This call is issued after initialising the ROM with service type to retrieve one byte from the ROM. A ROM should only respond to this if the ROM number in &F5 is 15-(ROM's physical number in &F4). The fetched byte should be returned in the Y register.

0F Vectors claimed. When a new filing system is initialised, it should, after writing its new vectors, issue this service call. This is to inform all the paged ROMs that there is a filing system change imminent.

10 SPOOL/EXEC file closure warning. This call is issued prior to closing the SPOOL and EXEC files when changing filing systems. This call is made primarily to allow any users of these files to tidy them up prior to their closure with the disappearance of the filing system. If a ROM responds to this call and returns with the accumulator zero, the SPOOL and EXEC files will not be closed. Care is necessary to prevent accidental access to files belonging to another (inactive) filing system.

11 Font implosion/explosion warning. Each time the fonts are exploded or imploded the high water mark will change. This call exists to inform languages that the high water mark has changed, and that the Y register contains the new value. This call should be used to get your precious data

out of the way before being destroyed by the character set. **BASIC**, it should be noted, ignores this warning, as it has no service entry.

12 Initialise filing system. If a program is transferring files from one filing system to another, it may need to have files open in more than one filing system. To do this, filing systems need to be re-activated before an access (all filing systems allow files to be open whilst inactive). This call exists to initialise a filing system without the fuss of issuing operating system commands. A filing system should respond to this call if the value in the Y register is its identification number. The filing system should initialise and restore all files still open when the filing system shut down.

FE Tube system post initialisation. This call is always issued after OSHWM has been set up. It is intended to allow the Tube system to explode the character set, or make other use of the main memory. On systems without the Tube, this call can be used for nefarious break trapping.

FF Tube system main initialisation. This call is issued only if the Tube hardware has been detected, and is called prior to message generation and filing system initialisation.

The language entry point

The language entry point is used to start up a language, and should only be entered with &01 in the accumulator on the BBC microcomputer. No return is expected from a language. The language entry point is always entered with a JMP instruction. The stack state on entry is undefined, and the stack pointer should be reinitialised.

The actual entry codes in the accumulator are:

0. No language present on break. No language ROM is entered this way, but the Tube language entry point might be.
1. Normal language start up.
2. Electron only. Request next byte of soft key expansion. Key number set by call with A=3. Byte out in Y.
3. Electron only. Request length of soft key expansion. Key number in Y. Length out in Y.

Vectored entry into paged ROMs

As many filing systems are paged ROM resident, a mechanism has been provided for changing vectors to point into paged ROMs.

Each vector has a number, n, such that the vector normally exists at location &0200+2*n. Any vector can be made to point into a sideways ROM by:

a) Making the main vector at location &0200+2*n point to &FF00+3*n. This is the operating system's entry point for processing extended vectors.
b) Ascertaining the extended vector space by issuing OSBYTE with A=&A8, X=&00, Y=&FF; this returns in X and Y the address of the extended vector space. Call this address V. (In OS1.20 V=&0D9F)
c) Setting the extended vector at location V+3*n to:
    a. address in ROM low byte
    b. address in ROM high byte ROM
    c. number (held in &F4)

Languages

A language on the BBC microcomputer is not necessarily a language at all, but could be any self contained machine code program that is independent of language. Examples are:

- Text editors and word processors
- Terminal emulators
- Teletext systems

Languages can only be started up by issuing an OSBYTE call with A=&8E, X=ROM number (except **BASIC** which is a special case). This is normally inconvenient and dependant on ROM position. A service entry should therefore be provided which uses the unrecognised command entry (service type 4) to recognise the language name as a command. Upon recognising the language name, the ROM should issue the OSBYTE call to properly start up the language.

When starting up a language the operating system does the following:

a) Records the ROM number to reselect the language on errors and soft breaks.

b) Displays the ROM name.
c) Leaves the error message pointer pointing to the copyright message or the version string if present.
d) If a second processor is active, copies the language across the Tube to the second processor and executes it there. If there is a relocation address it is taken account of during the transfer.
e) Enters the language at its language entry point.

When entered at the language entry point, a language should always set up the BRK vector to its error handler. Even the simplest language requires error handling, since just writing a character to the screen can cause an error when output is spooled. The error handler should output the error message, and continue execution within the language ROM at a suitable well defined point (such as the keyboard input prompt). The BRK vector does not need to be an extended vector, as the operating system automatically switches to the current language ROM on a BRK. When a BRK occurs, the currently active paged ROM number is recorded, and can be read with OSBYTE &BA. Service paged ROMs normally copy their error messages into RAM, so that the language does not have to read the error message from another ROM.

Languages must also enable interrupts, otherwise the operating system will fail to work.

Languages have the following workspace available:

&0000 - &008F	Page zero
&0400 - &07FF	Main language workspace
OSHWM - screen bottom	Program space

Filing systems

Filing systems are discussed fully in the Filing Systems chapter 21. This section covers filing systems and their relations with paged ROMs.

Initialising a filing system

Filing systems that are ROM resident can be initialised in three ways, all of which should be catered for in a filing system ROM:

1. An operating system command is issued naming the filing system. The filing system should watch for its selection command coming through a service call.
2. A service call &12 is issued with Y equal to the filing system number.
3. Auto-booting on system reset. If a filing system receives a service call &03, it should initialise if appropriate (see service call &03 details). Note that at this stage the language has not been initialised, so errors will be treated oddly. If Y=0 at this point, a boot file should be searched for if appropriate.

On initialisation a filing system should do the following:

a) Call OSFSC with A=6 (see filing systems section) to warn the old vector owner that the vectors are about to be redefined.
b) Set up the extended vectors, as previously specified.
c) Issue service call &0F, to warn all other paged ROMs of the vector change.
d) Restore all open files from the last activation of the filing system. Filing systems should be able to keep files open on 'hold' whilst inactive.

Other considerations for filing systems

If a filing system wishes to issue an error, it cannot do it in the normal way of executing a BRK instruction, because when the language comes to process the message, the message will be unavailable in another ROM. The usual way of issuing errors is to copy to location &0100 (the far end of the stack) a BRK instruction, the error number and error message, and then jump to location &0100.

The following workspace is reserved for filing systems:

Base page:

&A0-&A7	NMI workspace. Can only be used whilst NMI claimed.
&A8-&AF	Utilities area. Can only be used within an operating system command (can be a filing system command).
&B0-&BF	Filing system scratch space. Contents are not guaranteed to remain present from one call to another. Do not use for interrupt routines.
&C0-&CF	Filing system dedicated workspace. This is guaranteed to remain intact as long as the filing system is active, and as long as the absolute workspace is not claimed.

Main memory:

&0D00 - &0D5F	NMI code area. Filing systems using NMI code should copy the code to this space after claiming this space with service call &0C. If the NMI code must access the ROM, it should save the old ROM number to restore at the end of NMI service.
&0E00 - &**ssss**	Absolute workspace. The address **ssss** is set at reset by service call &01. Must be claimed before use by service call &0A.

ROM filing system software

ROM filing system (RFS) software that is resident in paged ROMs should contain header code to provide data which has been requested by service calls &0D and &0E. Each paged ROM should contain an end-of-ROM code (&2B '+') after the last block. Data should be formatted as specified in the ROM Filing System section, in Chapter 21.

A simple ROM filing system service code block could be:

```
.serve  CMP #&00        \ Is it a ROM initialising call?
        BNE ninit       \ No - branch
        PHA             \ Save the service call type
        TYA             \ Get logical ROM number
        EOR #&0F        \ Adjust it (do 15-x)
        CMP &F4         \ Compare with this ROM's number
        BCC notus       \ Number lose, this ROM already been done
        LDA #data MOD 256       \ Low byte of data address
        STA &F6         \ Location &F6 and &F7 reserved for this
        LDA #data DIV 256       \ High byte of data address
        STA &F7         \ Save high byte
        LDA &F4         \ This ROM's number
        EOR #&0F        \ Adjust it back (15-x)
        STA &F5         \ Pass over any higher non-filing system
                        \ ROMs
        PLA             \ Discard service type
        LDA #0          \ Set service type to no-operation
        RTS             \ Exit
.notus  PLA             \ Restore service type
.exit   RTS             \ Exit
.ninit  CMP #&0E        \ Is it a byte requests call?
        BNE exit        \ No - exit
        PHA             \ Save service type
        LDA &F5         \ Find the current ROM number
        EOR #&0F        \ Adjust it
        CMP &F4         \ Compare it with this ROM
        BNE notus       \ Not the same, must be for another ROM
        LDY #0          \ Prepare to get the data
        LDA (&F6),Y     \ Get the data
        TAY             \ In the Y register ready for exit
        INC &F6         \ Increment the address for next time
        BNE ninc7       \ No need to increment &F?
        INC &57
.ninc7  PLA             \ Discard service type
        LDA #0          \ Set service type to no-operation
        RTS             \ Exit
.data                   \ Data onward from here
```

21. Filing systems

Files

A file, to the BBC microcomputer, is a sequence of bytes. A file has a number of attributes: a load address, an execution address, a length, and, if 'open', a sequential pointer.

Every file is given a name when it is created. When a file is used it is identified by this file name.

When a file is opened for single byte access the filing system allocates the file a 'channel'. The channel can be considered to be a window into a file. A 'handle' is assigned to each channel to identify it internally. A file may be opened more than once for input, and so more than one channel can be associated with one file at any one time. When the file is closed and no further access is required then the channel is released and is no longer valid. The channel that was in use is now available to be assigned to another file if required. Any filing system can only have a limited number of open channels at one time and so there are only a limited number of handles available. A sequential pointer is maintained for each open file. The sequential pointer always points to the next byte to be read or written. In the disc filing system it is possible to change the value of this pointer to give random access within a file. Only serial access is available with the tape filing system.

Directories

On the disc and network filing systems, each file is resident in a directory. A file is identified by its directory followed by its name. Directories are separated from filenames by a full stop. For example, in 'A.NAME', 'A' is the directory name, and 'NAME' is the file name. For the Disc Filing System, the directory can only be a single character. In the Advanced Disc Filing System, directories can have up to 10 characters, the same as a filename. With ADFS, you may also have directories within directories, something not possible in DFS.

For convenience, the user may specify a 'current' directory, which is assumed if the directory name is omitted from the file identification.

The user may also specify a 'library' directory, which is similar to a 'current' directory. The library directory is searched for unrecognised commands unless a different directory is included in the command name.

Cycle numbers

The network and disc filing systems also use a 'cycle number' associated with the disc. The cycle number represents the number of times that a particular disc catalogue has been written to. It is of use in helping to identify a disc with greater reliability.

Filing systems

A filing system is a program that manages files on a particular storage medium. Filing systems are entered through vectors held in page 2 of memory.

Filing systems, upon selection, must provide a series of seven vectors, from location &212 to &21F. These point to the relevant routines within the filing system.

The filing system vectors are:

&212	FILEV	Operations on whole files.
&214	ARGSV	Adjust file arguments.
&216	BGETV	Get one byte from an open file.
&218	BPUTV	Put one byte to an open file.
&21A	GBPBV	Get/put a block of bytes to/from an open file.
&21C	FINDV	Open/close a file for byte access.
&21E	FSCV	Filing system control - various actions.

Filing systems are selected either during a reset, or by means of an operating system command; thus filing systems must also trap the command line interpreter to catch their selection command. Filing systems resident in paged ROMs are automatically informed by the operating system of any unrecognised command attempted.

Calls to the filing system vectors vary depending on the filing system in use. Refer to the User Guide or Advanced User Guide of the filing system in question, for details on these calls.

Filing systems and the Tube

Filing systems are required to communicate with the Tube system, and inform it of the following transactions:

- Write to second processor memory
- Read from second processor memory
- Start execution in the second processor from a given address

A filing system should only enter communication with the Tube system when the Tube is present (checked with OSBYTE &EA), and when the address required is not in the I/O processor. The BBC microcomputer system uses a 32 bit addressing system, and the I/O processor's memory is normally selected when the top 16 bits are &FFFF, other addresses being second processor memory.

When the Tube is present some machine code instructions will be present at location &406. If file addresses require data to be transferred over the Tube a call should be made to &406 with the following parameters:

X,Y point to a control block in memory, X low byte, Y high byte. The control block contains a four byte address.

A operation parameter, this can take the following values:
A=0 Initialise read of second processor memory
A=1 Initialise writing of second processor memory
A=4 Start execution in the second processor

Data is transferred to and from second processor memory over the Tube via a Tube register at location &FEE5. Thus to read second processor memory, a memory read is initialised with A=0, then successive reads are performed of location &FEE5.

The Cassette filing system

This filing system is provided as standard on all BBC microcomputers. It is very basic, and many entry points are not implemented, or are only partially implemented.

The cassette tape format is:

5 seconds of 2400Hz tone to synchronise the ACIA.

A header block, which is:

- 1 byte synchronisation. (&2A '*')
- Filename (1 to 10 characters)
- Filename terminator (&00)
- Load address of file, four bytes, low byte first.
- Execution address of file, four bytes, low byte first.
- Block number, two bytes, low byte first.
- Block length, two bytes, low byte first.
- Block flag, one byte:
    - Bit 0 Protection bit. The file can only be *RUN if this bit is set
    - Bit 6 Empty block bit. This block contains no data if this bit is set. An empty block is created when a file is opened for output and immediately closed again without BPUTting anything
    - Bit 7 Last block bit. This block is the last block of a file if this bit is set
- Four spare bytes (&00)
- Header Cyclic Redundancy Check (CRC), two bytes, high byte first.

A data block, which is:

- Data, 0 to 65535 bytes (the usual maximum is 256), as specified in the header.
- Data CRC, two bytes, high byte first.

The header CRC excludes the synchronisation byte.

A cyclic redundancy check value (CRC) is a number which can be calculated from a block of data. If the data is changed and another CRC performed the CRC value will probably be different. When the cassette filing system saves data onto tape it calculates the CRC for each block of data and includes it in the cassette format. When data is reloaded from cassette a CRC is calculated and compared with the CRC value stored with the data. If the two CRC values are not the same then the data has been corrupted.

Cyclic redundancy checking is sensitive enough to detect single bit errors, but robust enough so that multibit errors are unlikely to cancel out. Parity checking is used in some data transfer situations where the consistency of the data is being checked. Parity checking involves adding a bit to each byte of output. This bit is calculated so that the total number of set bits, including the parity bit, is consistently either odd or even. Whether odd or even is not important, as long as it is consistent. CRCs are usually used rather than byte-by-byte parity checking because if two bits are in error in a single byte, parity may not pick it up, though a CRC will.

CRCs may be calculated by using the following code, which calculates the CRC for a block of bytes of length 'lblk' and starting at 'data'.

On exit, H contains the CRC high byte and L contains the CRC low byte.

```
        LDA #0
        STA H           \ Initialise the CRC to zero
        STA L
        TAY             \ Initialise the data pointer
.nbyt   LDA H           \ Work data into CRC
        EOR data,Y
        STA H
        LDX #8          \ Perform polynomial recycling
.loop   LDA H           \ Loop is performed 8 times, once for bit
        ROL A           \ Test if a bit is being cycled out
        BCC b7z
        LDA H           \ Yes, add it back in *~8~5
        EOR #8
        STA H
        LDA L
        EOR #&10
        STA L
.b7z    ROL L           \ Always, rotate whole CRC left one bit
        ROL H
        DEX
        BNE loop        \Do once for each bit
        INY             \Point to next data byte
        CPY #lblk       \All done yet?
        BNE nbyt
        RTS             \All done- H=CRC Hi, L=CRC Lo
```

ROM filing system

The ROM filing system is standard on all BBC microcomputers with 1.0 operating system, or later. The ROM filing system uses data stored in paged ROM or in serially accessed ROMs associated with the speech processor. The socket on the left hand side of the keyboard is designed to accommodate ROM packs containing speech ROM devices; these require the presence of a speech system upgrade.

This filing system is the same as the cassette filing system in every way, except:

- OSFILE Save is meaningless.
- OSBPUT Writing to ROM is not sensible.
- OSFIND Opening for output is not possible. The handle given for input is 3.

The internal format for ROMs is the same as the tape format, with the following exceptions:

&2B ('+') is used as an end-of-ROM marker. Files can span over an end of a ROM marker, but care should be taken to ensure that the right ROM is read at the right time (i.e. the next sequential block in the file must be in the next ROM to be read).

In the ROM filing system, the whole header may be replaced by a single character (&43 '#') for all bar the first and last blocks.

If the header is abbreviated in this way, it is assumed to mean that the header is unchanged from the last block, except for the block number.

In the ROM filing system, the 'four spare bytes' in the cassette header block are used to contain the address of the byte after the end of the file, enabling file searches to be a lot faster. Fast searching is used by the ROM filing system by default. Full CRC checking of each block during a `*CAT` can be enabled by issuing a `*OPT 1,2` command.

ROM software may be resident in standard paged ROMs, or, if the speech chips are fitted, in a PHROM (PHrase Read Only Memory). Storing data in paged ROMs is covered in Chapter 20. Data stored in PHROMs, is recognised as data, as opposed to speech, by an identification sequence.

This has the following format:

00	ignored
01	&00
02	&28 '('
03	&43 'C'
04	&29 ')'
05	
...	ignored
3D	
3E	address of data in internal format for the speech chip indirection command.

Disc filing system

The disc filing system is not provided as standard with the BBC microcomputer. Extra hardware and software are required to operate disc drives. The disc filing system is complete except in some small areas. It differs from the standard filing system protocol in the following ways:

- A restricted form of file attributes is used: the four bytes are compressed into one bit. This bit is referred to as the 'lock' bit. A file is locked if either of the 'you cannot write' or 'you cannot delete' bits is set. If either of these attributes is set then both attributes are set.
- File types 0 or 2 are never returned, instead a 'File not found' error is issued.
- The device identity is the drive number and is one character long.
- Directory names are one ASCII character.
- File names are up to 7 ASCII characters long. The media title is the disc title, and is up to twelve characters long.

Advanced disc filing system

This updated version of the standard disc filing system is not supplied as standard with the BBC Micro, and when fitted, requires a 1770/2 controller to use ADFS floppy discs, although you may use an 8271 controller if you only intend to use the

ADFS to access Compact Flash hard drives. In this instance, you may also have no disc controller fitted at all, and the CF system will still work correctly.

The main differences to the Disc filing system is as follows:

- Filenames can be 10 characters long
- Directory names can be 10 characters long, and can be themselves placed in other directories
- Drive numbers are dependent on system configuration, but are usually 0-1 for floppy only systems, and 0-3 for CF hard drives, and 4-5 for floppies, when the CF system is fitted.

Econet filing system

The net filing system is not provided as standard with the BBC microcomputer. Extra hardware and software are required. The networked filing system has the following characteristics:

- File attributes are complete. The second and third bytes of the attribute block are used to store the date when the file was created.
- The device identity is not applicable, and its length is zero.
- Directory names are one to ten ASCII characters long. File names are one to ten ASCII characters long.
- The media title is the disc title, and is up to sixteen characters long.

RetroClinic RamFS

This filing system is for use with the RetroClinic DataCentre system. It emulates 4 DFS 80 track floppy drives, as well as allowing access to USB flash drives, and a 64K NVRAM on the main DataCentre PCB. See Appendix K for more info on DataCentre.

22. Using printers

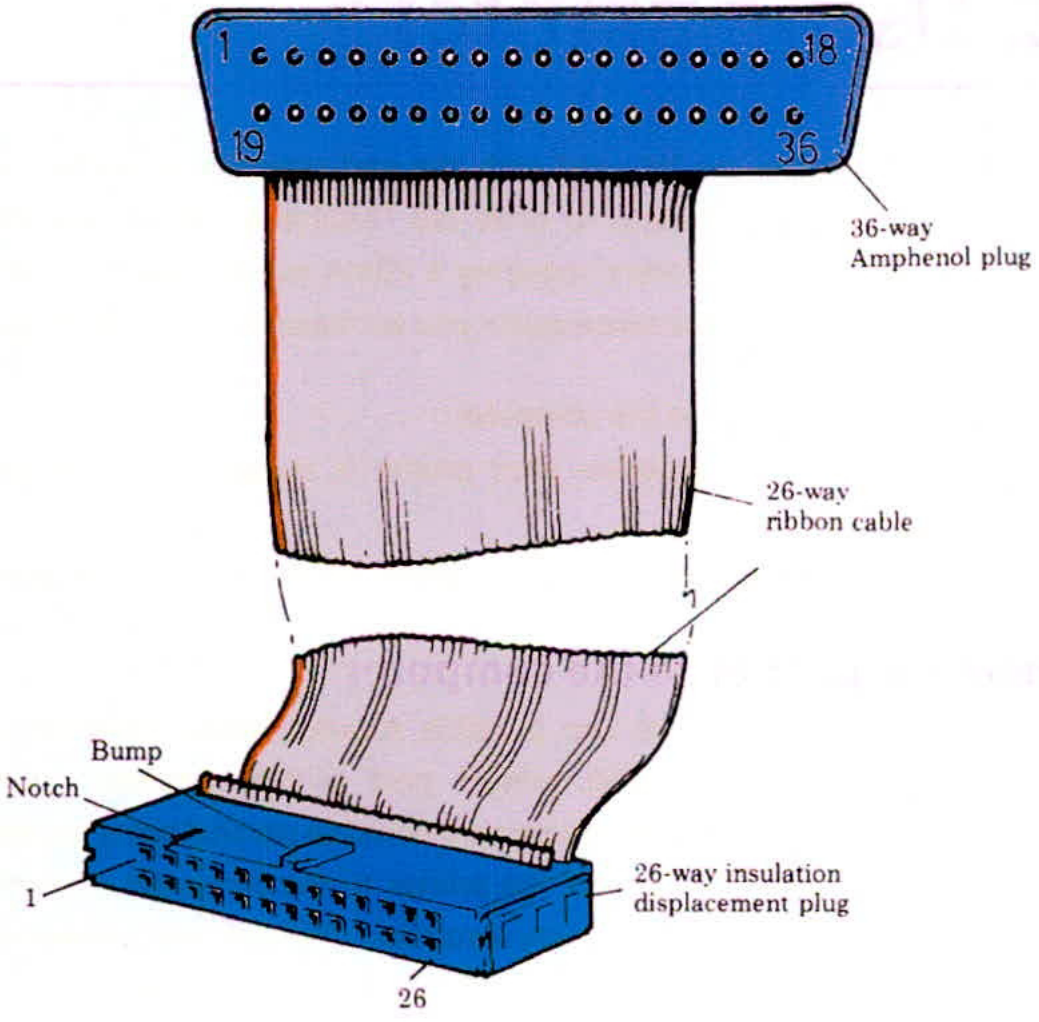
The BBC Microcomputer can be used with the vast majority of printers available today. It is not possible, though, to drive old 'Teletypes' which operate at ten characters per second, or printers requiring a 20mA current loop connection. To use a printer with your BBC Microcomputer you will need to do three things:

- Connect the printer to the computer.
- Tell the computer whether your printer is plugged into the parallel or serial printer port.
- Tell the computer to copy everything sent to the screen to the printer.

Connect the printer to the computer

As the above implies there are two possible sockets (ports) to which you can connect the printer. The parallel printer port is often called a Centronics compatible port. Printers that need to be connected to parallel ports usually have a 36-way Amphenol socket. To connect the printer to the computer you will need a cable made up as follows from a 26-way insulation displacement connector, about one metre of 26-way ribbon cable and one 36-way Amphenol plug.

The 26-way cable should be inserted into the 36-way plug so that pins 1 to 14 and 19 to 32 are connected. The cable should be inserted into the 26-way plug so that pin 1 of the Amphenol 36-way plug is connected to pin 1 of the 26-way insulation displacement plug.



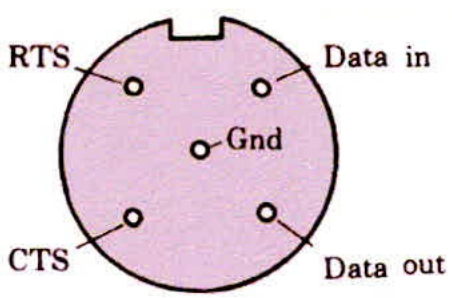
A parallel printer cable

Both plugs viewed looking at the pins on the underside of the plug body.

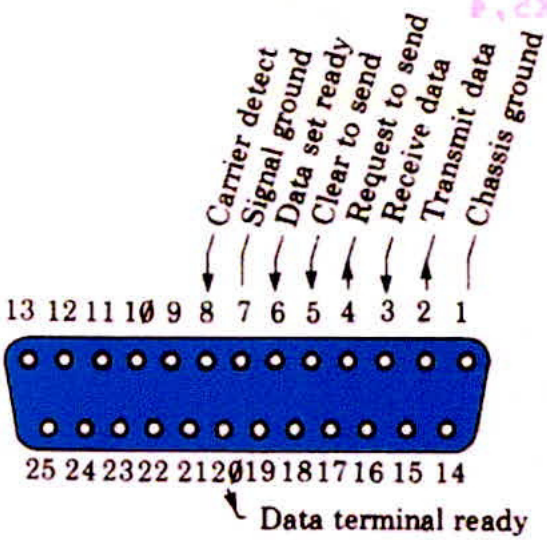
Parallel printer connections

Name	36-way plug pin number	26-way plug pin number
Strobe	1	1
Data 0	2	3
Data 1	3	5
Data 2	4	7
Data 3	5	9
Data 4	6	11
Data 5	7	13
Data 6	8	15
Data 7	9	17
Acknowledge	10	19
Ground	19 to 32	All even numbers (except-26)

The ***serial printer port*** is often referred to as an RS232 or V24 port. The standard connector plug is a Cannon 25-way D type and there is also a standard for the connections to the plug. Predictably however, different manufacturers disagree on the interpretation of the standard! The serial port on the BBC Microcomputer emerges via a 5-pin DIN plug and it will normally be possible to drive a serial printer with just three wires. The views of the two plugs are shown below from the outside of the case and looking into the respective sockets.



5-way DIN plug
for computer



25-way D type plug for printer

Proceed as follows:

- Connect a wire between GND and pin 7.
- Connect another wire between data out and pin 3.
- The position of the third wire varies from printer to printer.
    - For Qume printers connect CTS to pin 20.
    - If that doesn't work then try CTS to pin 4.

If you require a more complete RS232 connection try the following:

- Data in to pin 2 (transmit data).
- Data out to pin 3 (receive data).
- CTS to pin 4 (clear to send).
- RTS to pin 5 (request to send).
- 5 GND to pin 7 (signal ground).

Telling the computer whether you are using a serial or parallel printer

`*FX5` is used to inform the computer whether it is to send printer output to the parallel or serial port. Type:

`*FX5,1`	if you are using a parallel printer
`*FX5,2`	if you are using a serial printer
`*FX5,4`	if you are using the Econet printer

The computer defaults to a parallel printer, unless you tell it otherwise

If you have selected a serial printer then you will have to set the baud rate to match that on the printer. The default setting is 9600 baud but many printers run at other rates. Select one of:

`*FX 8,1`	75 baud
`*FX 8,2`	150 baud
`*FX 8,3`	300 baud
`*FX 8,4`	1200 baud
`*FX 8,5`	2400 baud
`*FX 8,6`	4800 baud
`*FX 8,7`	9600 baud
`*FX 8,8`	19200 baud (this rate is not guaranteed)

Telling the computer to copy everything to the printer

This is often called turning the printer on – but, of course, we are not referring to switching the printer on but to enabling and disabling its printing.

To turn it on type **<CTRL>** **<B>** and to turn it off type **<CTRL>** **<C>**.

In a **BASIC** program `VDU2` will turn the printer on and `VDU3` will turn it off.

Note that all output will now appear on the screen and printer. To send output to the printer only you can use `*FX3`.

For disc systems, `*TYPE filename` will print out a data file.

Characters not sent to the printer

Some printers automatically move the paper up one line when they receive a carriage return. Since the computer also moves the paper up one line (by using a line feed), the paper may move up two lines instead of one. It is possible to tell the computer not to send any particular character to the printer. The most common requirement is to ignore line feed which has an ASCII code of 10.

`*FX6,10` will set it to ignore line feeds. `*FX6` should be set after `*FX5` has been used to select the printer type. The default is to ignore line feeds.

The complete set-up sequence for a serial printer running at 1200 baud and which does not require line feeds would be:

`*FX 5,2`	Serial printer.
`*FX 8,4`	1200 baud.
`*FX 6,10`	No line feeds (default so it may be omitted).

A few more detailed points:

Data is transmitted using one start bit, eight data bits and one stop bit.

Receive baud rates may be the same as, or may differ from, transmit baud rates. Receive baud rates are set by `*FX7`.

The user may supply his or her own specialist printer driver routine. The address of the user routine should be placed at location &222 and the user defined routine can be selected as the printer output routine with `*FX5,3`.

If the user intercepts the operating system write character routine (OSWRCH) then all the VDU control codes must be dealt with. When a **BASIC** program executes `DRAW 10,10` a string of six bytes is sent to the VDU driver via OSWRCH. In this case the bytes would be `25,5,10,0,10,0` so beware!

USB Printers

It is possible to connect a USB printer up to the BBC Micro, using the RetroClinic DataCentre. Any USB printer will need an appropriate driver written for it, but the command set for sending data is documented in the Vinculum Datasheet, which accompanies the DataCentre kit when purchased.

23. The Hardware

Most users of the BBC microcomputer will be familiar with **BASIC** programs, but from **BASIC** the hardware is virtually invisible. Commands are provided to deal with output to the screen, input from the keyboard and analogue to digital converter, plus all of the other hardware. The same applies to machine code to a large extent through the use of OSBYTES, OSWORDS and other operating system commands. However, a much more detailed understanding of the hardware and how it can be controlled from machine code programs is very useful and allows certain features to be implemented which would have been impossible in **BASIC**.

The hardware section of this book satisfies the requirements of two types of people; those who wish to use the hardware features already present on the computer, and those who wish to add their own hardware to the computer. All of the standard hardware features available on the BBC microcomputer are therefore outlined in detail from a programmer's point of view. Wherever possible, it is better to use operating system routes for controlling the hardware. These are very powerful and will be referred to whenever relevant. In certain specialised cases, it is necessary to directly access hardware, but even in such cases, OSBYTES &92-&97 should be used. This will ensure that the software will still operate on machines fitted with a Tube processor. For those who wish to add their own hardware, full details on using the USER port and 1MHz BUS are supplied.

The hardware on the BBC microcomputer consists of a large quantity of integrated circuits, resistors, capacitors, transistors and various other electronic components. All of these are shown on the full circuit diagram that comes in the pouch on the back cover of the folder if this book. In order to help those who are not familiar with the general layout of a computer circuit and the devices attached to it, the rest of this introduction is devoted to analysing the hardware as a series of discrete blocks interconnected by a series of system buses.

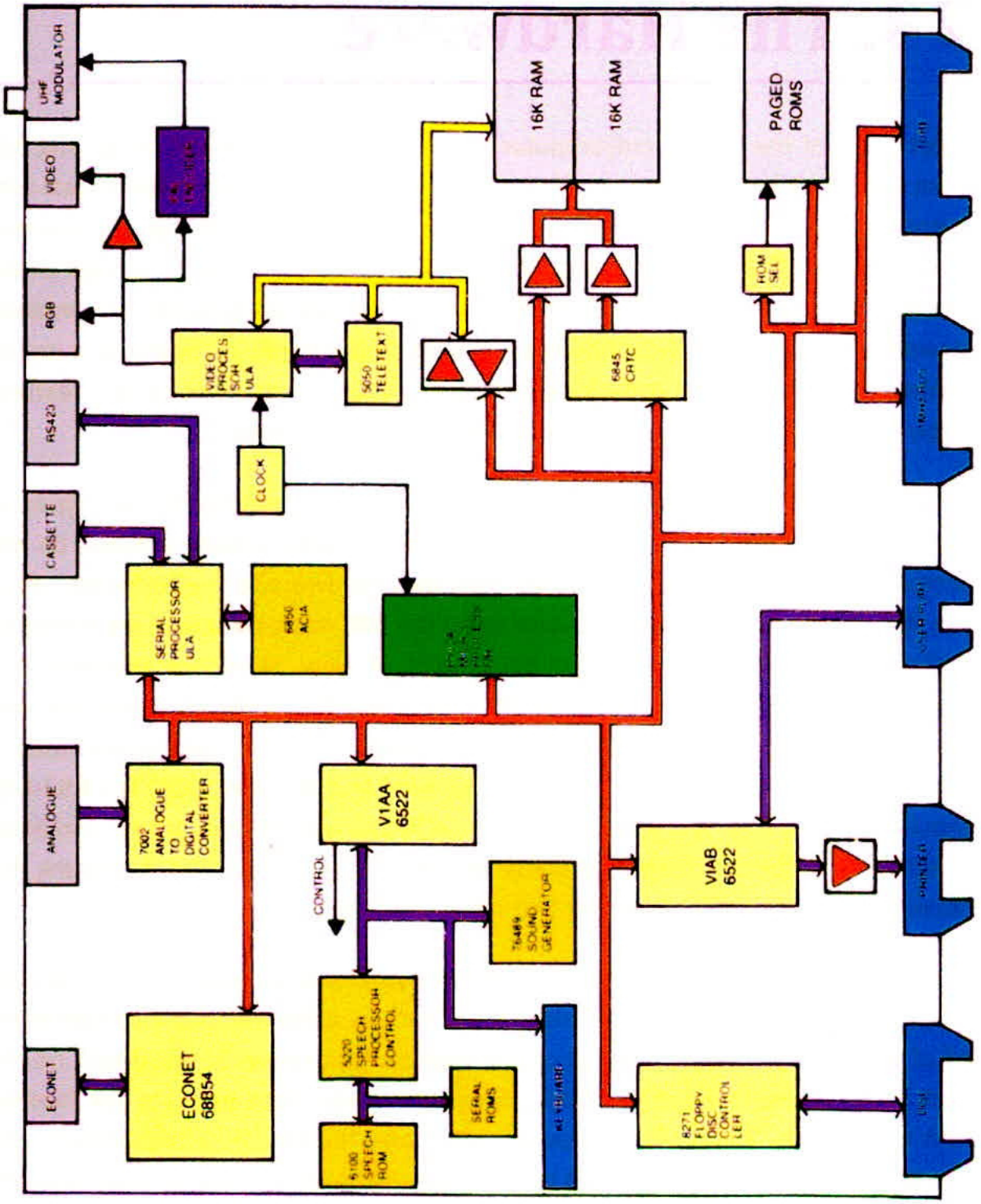


Figure 23.1 - The system block diagram

At the centre of the system is the 6502A central processing unit (CPU). This is the chip which executes all of the programs including **BASIC**. It is connected to the rest of the system via three buses. These are the data bus, the address bus and the

control bus. For clarity on the diagram, these buses are all compressed into one which is represented by the red lines terminated with arrows at each major block.

A *bus* is simply a number of electrical links connected in parallel to several devices. Normally one of these devices is talking to another device on the bus. The communication protocols which enable this transfer of data to take place are set up by the control, address and data buses. In the case of the address bus, there are 16 separate lines which allow 65536 different combinations of 1's and 0's. The maximum amount of directly addressable memory on a 6502 is therefore 65536 bytes. The data bus consists of 8 lines, one for each bit of a byte. Any number between 0 and &FF (255) can be transferred across the data bus. Communication between the peripherals, memory and the CPU occurs over the data bus. The CPU can either send out a byte or receive a byte. The data bus is therefore called a *bidirectional* bus because data flows in any one of two directions. The address bus is unidirectional since the 6502 provides, but cannot receive addresses. Note that some address buffers are included in the video circuit to allow either the 6845 or 6502 to provide addresses for system random access memory (RAM).

In order to control the direction of data flow on the data bus, a read or write signal is provided by the control bus. Hardware connected to the system can thereby determine whether it is being sent data or is meant to send data back to the CPU. The other major control bus functions are those of providing a clock, interrupts and resets. The clock signal keeps all of the chips running together at the same rate. The RESET line allows all hardware to be initialised to some predefined state after a reset. An interrupt is a signal sent from a peripheral to the 6502 requesting the 6502 to look at that peripheral. Two forms of interrupt are provided. One of these is the interrupt request (IRQ) which the 6502 can ignore under software control. The other in the non-maskable interrupt (NMI) which can never be ignored.

When power is first applied to the system, a reset is generated to ensure that all devices start up in their reset states. The 6502 then starts to get instructions from the MOS ROM. These instructions tell the 6502 what it should do next. A variety of different instructions exist on the 6502. The basic functions available are reading or writing data to memory or an input/output device and performing arithmetic and logical operations on the data. Once the MOS (machine operating system) program is entered, this piece of software gains full control of the system.

SHEILA and the system hardware

All of the main blocks connected to the 6502 in the block diagram, together form the system hardware. In 6502 systems, the hardware is *memory mapped* which means that any hardware device registers appear in the main memory address space. Page &FE (the 256 bytes of memory starting at &FE00) is reserved especially for the system hardware in the BBC microcomputer. The special name of 'SHEILA' has been assigned to this page of memory. Two other special pages are &FC (called 'FRED') and &FD (called 'JIM'). FRED and JIM are concerned with external user hardware attached to the one megahertz bus. They are dealt with later in this chapter, in the section on the one megahertz bus.

In the following sections, all of the devices attached to Sheila are described in detail. The table below shows the memory map of Sheila, the function of the devices attached to it.

SHEILA Address (offset from&FE00)	Integrated circuit	Description
&00-&07	6845 CRTC	Video controller
&08-&0F	6850 ACIA	Serial controller
&10-&1F	Serial ULA	Serial system chip
&20-&2F	Video ULA	Video system chip
&30-&3F	74LS161	Paged ROM selector
&40-&5F	6522 VIA	SYSTEM VIA
&60-&7F	6522 VIA	USER VIA
&80-&9F	8271 / 1770 FDC	Floppy disc controller
&A0-&BF	68B54 ADLC	ECONET controller
&C0-&DF	uPD7002	Analogue to digital converter
&E0-&FF	Tube ULA	Tube system interface

NOTE: Some Sheila addresses are not normally used. This is because the same devices appear at several different Sheila addresses. For example, the paged ROM select register is normally addressed at location &30, but (on a Model B) it could equally well be addressed at any one of the fifteen other locations &31-&3F.

The 6845 CRTC

Sheila address &00-&07

The 6845 cathode ray tube controller chip (CRTC) forms the heart of the BBC Micro's video display circuitry. Its major function is that of displaying the video data in memory on a raster scan display device (a television or monitor). As an extra bonus, the 6845 also refreshes all of the random access memory so that the data stored there is not lost. This refreshing process is inherent in the sequential nature of accessing memory for the video display. The 6845 does not interfere with processor access to the memory since the processor and 6845 operate on alternate phases of the system clock. The 6845 is responsible for producing the correct format on the display device, positioning the cursor, performing interlace if it is required and monitoring the light pen input. Other video processing functions involving colour and teletext are dealt with in conjunction with other sections in Sheila.

Inside, the 6845 is a very powerful and complex VLSI chip. From a user's point of view it is useful to know how to define a specialised screen layout, and how the screen layouts (modes) have been defined by Acorn. A generalised overview of the 6845 is therefore given first, followed by the values in each of the registers in the various modes. This chapter ends with a general summary table which describes the functions of the various registers. Appendix F contains diagrams illustrating all of the screen modes in a very concise and easily referred to format.

Programming the 6845

The 6845 possesses 18 internal registers, 14 of which are write only (R0-R13), 2 of which are read and write (R14-R15) and 2 of which are read only (R16-R17). In order to gain access to any of these registers, the register address must be written into the 6845 address register. This is situated at Sheila address &00.

Having written a 5 bit number into the address register, the selected internal register may be written to or read from at Sheila address &01.

The best way of programming the 6845 is by using the `VDU23` command. For example `VDU23,0,R,V,0,0,0,0,0,0` will put the value V into register R. In **BASIC** programs it can be shortened by using semicolons instead of commas. A

semicolon causes a 2 byte word to be included in the `VDU` command. For example `VDU23;R,V;0;0;0` has the same effect as the first example.

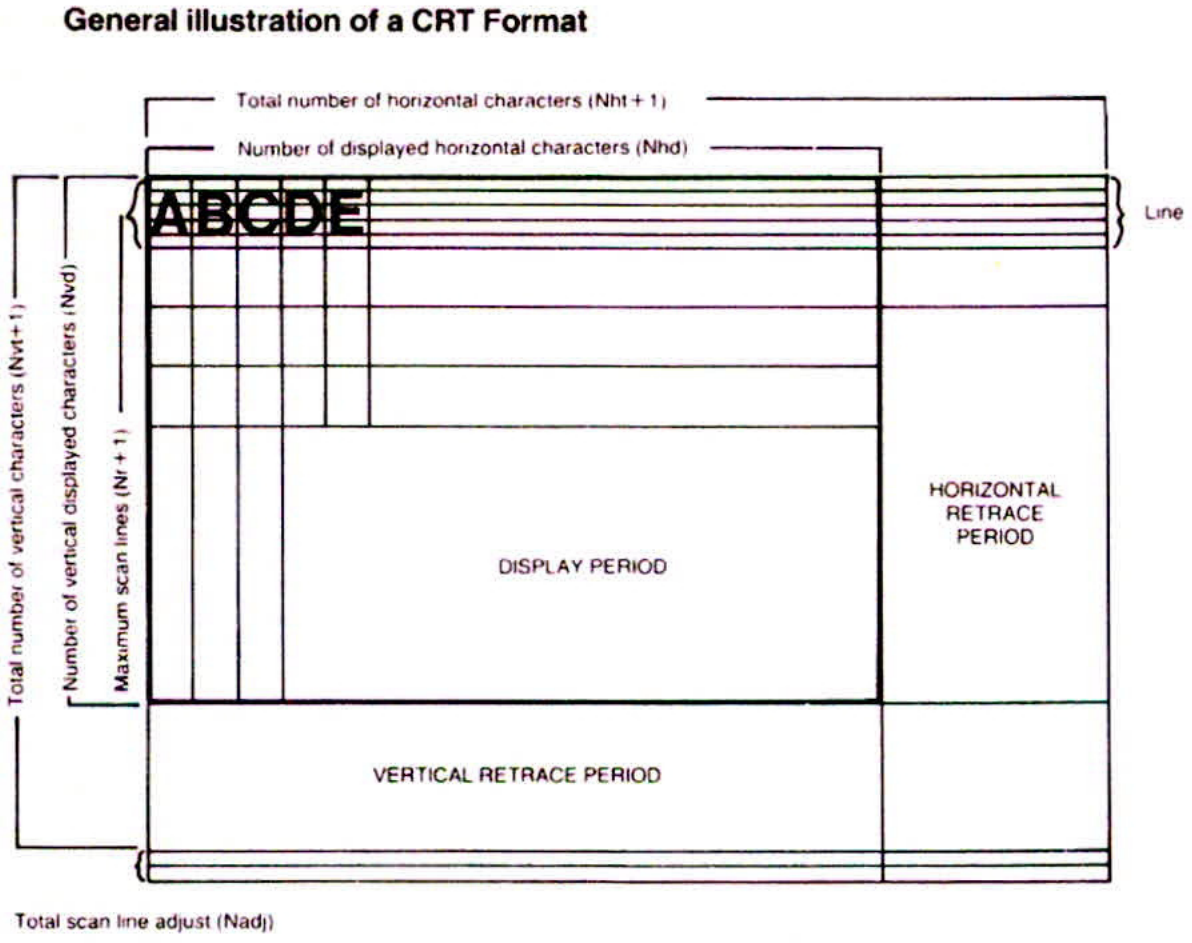


Figure 23.2 – Illustration of a general screen format

The Horizontal timing registers

The horizontal timing registers define all of the horizontal timing for the screen layout. The point of reference for these registers is the left most displayed character position. The registers are programmed in 'character time units' relative to the reference point.

Horizontal total register (R0)

This 8 bit write only register determines the horizontal sync. frequency. It should be programmed with the total number of displayed plus non-displayed character time units across the screen minus one (Nht in Figure 23.2).

Note that the number of displayed characters is not necessarily the same as the number of characters per line. This is because of the variable number of bits attributed to each pixel, depending upon the number of colours available. The table for R1 contents illustrates this.

Mode	0	1	2	3	4	5	6	7
R0	127	127	127	127	63	63	63	63

Horizontal displayed register (R1)

This 8 bit write only register determines the number of displayed characters per horizontal line (Nhd in Figure 23.2).

Mode	0	1	2	3	4	5	6	7
No. of CHRS as seen by 6845 (Nhd)	80	80	80	80	40	40	40	40
No. of CHRS as seen on the screen	80	40	20	80	40	20	40	40
No. of bits used to store colour information	1	2	4	1	1	2	1	1

Horizontal sync position register (R2)

This 8 bit write only register determines the horizontal sync. pulse position on the horizontal line. The specification is in terms of character widths from the left hand side of the screen.

Mode	0	1	2	3	4	5	6	7
R2	98	98	98	98	49	49	49	51

Increasing the value of this register pushes the entire screen left whilst decrementing it pushes the whole screen right.

The sync width register (R3)

This 8 bit write only register defines both the horizontal and the vertical sync. pulse times.

Horizontal sync pulse width

The lower 4 bits contain the horizontal sync. pulse width in number of characters. Any number between 1 and 15 can be programmed, but 0 is not valid. It is however not advisable to change this register since most monitors and televisions require the standard sync. width to operate properly.

Mode	0	1	2	3	4	5	6	7
Lower bits of R3	8	8	8	8	4	4	4	4

Vertical sync pulse width

The upper 4 bits contain the number of scan line times for the vertical sync. pulse. This is set to 2 in all modes.

The Vertical timing registers

The point of reference for vertical registers is the top displayed character position. Vertical registers are programmed in character row times or scan line times.

Vertical total register (R4)

The vertical sync. frequency is determined by both R4 and R5. In order to obtain an exact 50Hz or 60Hz vertical refresh rate, the required number of character line times is usually an integer plus a fraction. The integer number of character lines minus one (Nvt in Figure 23.2) is programmed into this 7 bit write only register.

Mode	0	1	2	3	4	5	6	7
R4	38	38	38	30	38	38	30	30

Vertical total adjust register (R5)

This 5 bit write only register is programmed with the fraction for use in conjunction with register R4. It is programmed with a number of scan lines (Nadj in Figure 23.2). It can be varied slightly in conjunction with R4 to move the whole display area up or down a little on the screen. It is usually set to 0 except when using modes 3,6 and 7

in which it is set to 2. *TV (OSBYTE &90) controls the vertical positioning of a display on the screen. Refer to the OSBYTE section for more details.

Vertical displayed register (R6)

This 7 bit write only register determines the number of displayed Character rows (Nvd) on the CRT screen and is programmed in character row times.

Mode	0	1	2	3	4	5	6	7
Character lines	32	32	32	25	32	32	25	25

Vertical sync position (R7)

This 7 bit write only register determines the vertical sync position with respect to the reference. It is programmed in character row times.

Mode	0	1	2	3	4	5	6	7
Sync position	34	34	34	27	34	34	27	27

Interlace and delay register (R8)

This 6 bit write only register controls the raster scan mode and cursor/display delay. The interlace options are:

Interlace modes (bits 0,1)

Interlace mode register		
Bit 1	Bit 0	Description
0	0	Normal (non-interlaced) sync mode (figure 23.3a)
1	0	Normal (non-interlaced) sync mode (figure 23.3a)
0	1	Interlace sync mode (figure 23.3b)
1	1	Interlace sync and video (figure 23.3c)

All BBC microcomputer screen modes are interlaced sync only except for mode 7 which is interlaced sync and video. The default values can easily be changed using *TV (*FX144) followed by a 0 to turn interlacing on or a 1 to turn interlacing off.

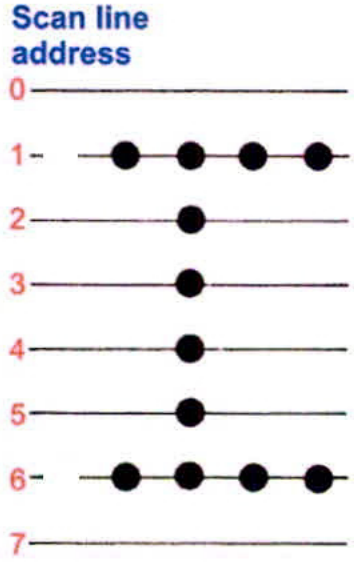


Figure 23.3a
Normal Sync

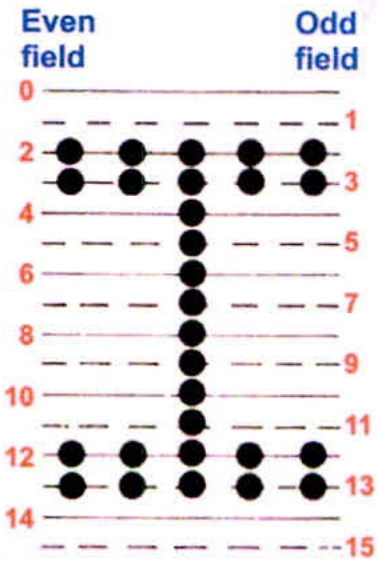


Figure 23.3b
Interlace Sync

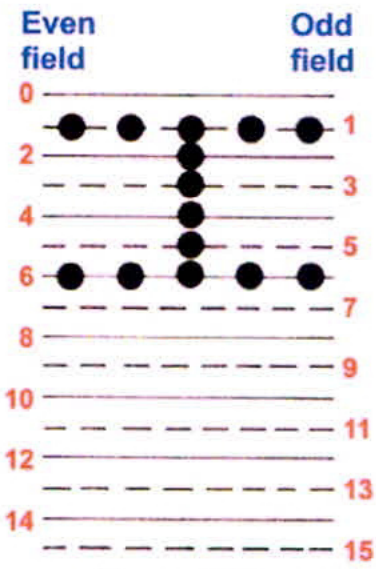


Figure 23.3c
Interlace Sync & video

Display blanking delay (bits 4,5)

Bits 4 and 5 control the display blanking signal. This signal must be enabled for all of the character output period and is used to take account of the time to transfer data from memory to the video output circuitry. No delay is required in modes 0-6, but a one character delay is required in mode 7 because the SAA5050 character generator is used.

Display blanking delay

Bit 5	Bit 4	Description
0	0	No delay
0	1	One character delay
1	0	Two character delay
1	1	Disable video output

Cursor blanking delay (bits 6,7)

Bits 6 and 7 control the cursor blanking signal. This signal must be enabled at the exact time when a cursor should appear on the screen. No delay is required in modes 0-6, but a two character delay is required in mode 7.

Cursor enable signal

Bit 7	Bit 6	Description
0	0	No delay
0	1	One character delay
1	0	Two character delay
1	1	Disable cursor output

Scan lines per character (R9)

This 5 bit write only register determines the number of scan lines per character row including spacing. The programmed value is one less than the total number of output scan lines.

Mode	0	1	2	3	4	5	6	7
Scans	7	7	7	9	7	7	9	18

THE CURSOR

It is possible to program a cursor to appear at any character position (defined by R14 and R15). Its blink rate can be set to 16 or 32 times the field period of 20 ins. Optional non-blink and non-display (i.e. no cursor on the screen) modes can also be selected. Its height in number of lines and its vertical position in a character slot can be defined as well.

The cursor start register (R10)

This 7 bit write only register controls the cursor format (see Figure 23.4). Bit 7 is not used. Bit 6 enables or disables the blink feature. Bit 5 is the blink timing control bit. When bit 5=0, blink frequency = 1/16th of field rate. When bit 5=1, blink frequency = 1/32nd of the field rate. The cursor start line is set by the lower five bits.

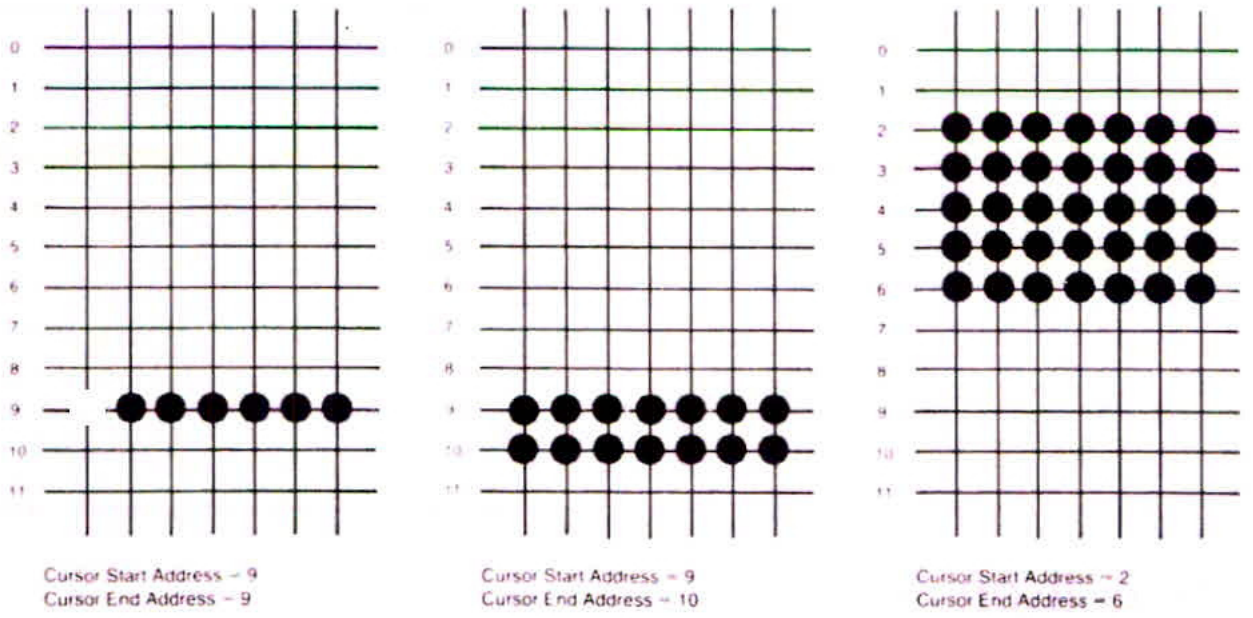


Figure 23.4 – Cursor layout examples

The cursor end register (R11)

This 5 bit write only register sets the cursor end scan line (see diagram).

Mode	0	1	2	3	4	5	6	7
Cursor end	8	8	8	9	8	8	9	19

Cursor position register (R14 and R15)

This 14 bit read/write register stores the current cursor location. It consists of 8 low order (R15) and six high order (R14) bits.

LIGHT PENS

A typical light pen consists of a small light sensitive device fixed to the end of a pen shaped holder. The sensor picks up the light given out from the monitor screen and sends an electronic signal into the micro. This LPSTB (light pen strobe) signal can be decoded (because the screen is scanned on a raster basis) and the position of the pen head determined. Lightpens of this type do not work on more modern LCD or Plasma displays, as these do not "scan" in the same way as a raster CRT.

Light pens can be used for a multitude of tasks such as drawing, 'painting', designing layouts, playing games etc., but their use in many applications is limited by the resolution. The reason for this is that a fairly large area of screen (i.e. Perhaps 0.5cm x 0.5cm) is usually required to provide sufficient light to operate the pen. The maximum resolution for defining the position of the light pen is therefore a patch on the screen of this size, so accurate line drawings are impossible. The position of the light pen is stored to the nearest character position, so this limits the resolution to a character cell.

Light pen position register (R16 and R17)

This 14 bit read only register is used to store the location of a light pen sensor placed in front of the screen. The register is modified whenever the LPSTB signal is pulsed high.

Constructing a light pen

The light pen hardware must produce a positive going TTL pulse whenever the display scan position is under the sensor. The light pen position will then be stored in the light pen register R16, R17. Note that slow light pen response will require a delay factor to be subtracted from R16, R17 to produce the correct light pen position.

Luckily, there are small light sensitive devices available which provide a direct TTL logic level output. If one of these is fixed to the end of an empty pen and connected to the light pen input on the rear of the BBC microcomputer, an operational light pen can be constructed. The connections for such a pen are illustrated in figure 23.5. A special photosensor called a 'Sweet spot' is available from RS Components, Farnell or similar.

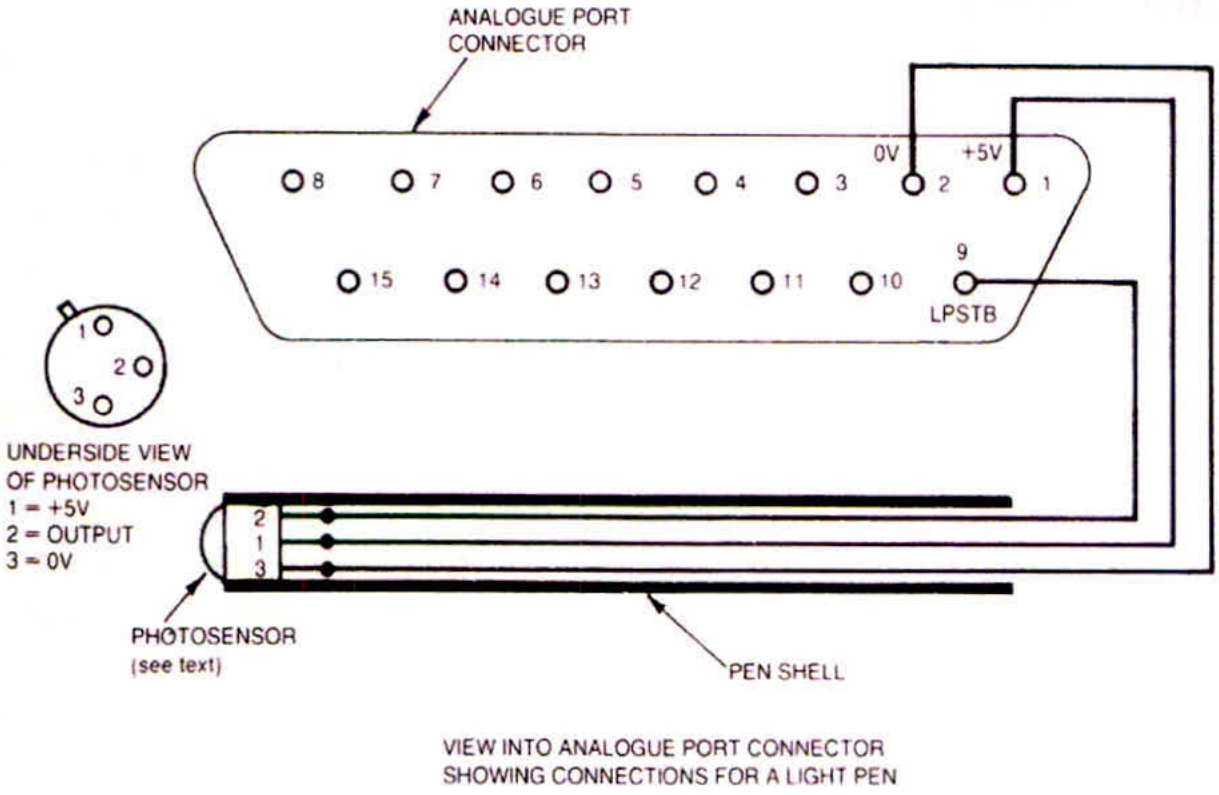


Figure 23.5 – Light pen circuit

Light pen software

In order to take account of the different screen start addresses for the various modes, a further correction factor must be subtracted from the contents of the light pen register. These correction factors are:

Mode	Correction factor
0	&0606 (1542)
1	&0606 (1542)
2	&0606 (1542)
3	&0806 (2054)
4	&0B04 (2820)
5	&0B04 (2820)
6	&0C04 (3076)
7	&2808 (10248)

The light pen position in terms of x,y co-ordinates is given by:

y = (L.P register-correction) DIV number of characters per line
x = (L.P register-correction) MOD number of characters per line

This x value will be in terms of 6845 characters and will have to be modified by multiplying by:

$$\frac{\text{Number of characters per line on screen}}{\text{Number of characters as seen by 6845}}$$

e.g. for mode 2 = 20/80 = 1/4 The resolutions are therefore:

Modes 0, 3, 4, 6 and 7	single displayed character
Modes 1 and 5	half of a displayed character
Mode 2	quarter of a displayed character

NOTE: The screen should be cleared before using a light pen and not scrolled whilst the pen is in use. If it is scrolled, the position of the start of the screen will have to be taken into account as well.

Displayed screen start address register (R12,R13)

This 14 bit write only register determines the location in memory which corresponds to the upper left hand character displayed on the screen. R13 (8 bits) is the low order address and R12 (6 bits) is the high order address. It can often be useful to know what the current contents of this register are. Unfortunately, being a write only register it is not possible to read the value directly. However, OSBYTE &A0 can be used to get these parameters from the operating system workspace in page &03. The start of screen address is stored in locations &350 and &351. Call OSBYTE &A0 with X=&30. The contents of &350 will be returned in the X register and the contents of &351 will be returned in the Y register.

Note that the actual screen start address must be divided by 8 before being sent to R12,R13 because there are 8 lines per character (modes 0-6). In mode 7 a rather more complex correction has to be applied. See the section on mode 7 scrolling below.

The ability to define the start of the screen to be anywhere in memory is very useful because it allows fast scrolling of the screen up, down, left and right. Provided that the start address is inside the screen memory of the mode being used, a hardware wrap around feature will also operate. Characters which would

have scrolled off the top of the screen will therefore reappear at the bottom. The wrap around circuit simply detects whenever the 6845 tries to get video data from a ROM (an address above &7FFF), and adds an offset to that address. This has the effect of bringing the address back inside the video RAM. Since the screen sizes are different in the various modes, 2 bits on the SYSTEM VIA are used to define the length of the hardware scrolled screen.

Hardware scrolling

Scrolling the screen fast in any direction can be of immense use in a large number of applications. Text can be scrolled in word processing applications, landscapes can be made to rush by in a horizontal direction (see games such as Acornsoft Planetoid). If it were not for the hardware scroll feature, it would be necessary to move every byte on the screen to perform a scroll. This is very time consuming for 20000 bytes and therefore slow. In order to make effective use of the hardware scrolling facilities available, it is necessary to understand both the advantages and the limitations which are imposed.

Modes 0-6 will now be analysed in detail followed by mode 7 which is slightly different.

Modes 0-6 vertical scrolling

In order to move the screen position upwards by one character line, it is necessary to increment the current start address register (R12,R13) by the number of characters per line. Remember that these are characters as produced by the 6845 and not as seen on the screen. There are 80 6845 characters per line in modes 0-3 and 40 characters per line in modes 4,5 and 6. The screen can be scrolled downwards by decrementing the screen start address register by the number of characters per line. Note that you should not normally allow the screen start address register to contain a value less than the *official* screen start address or greater than the *official* screen end address in the mode being used. If this occurs then areas of the main system memory will be displayed directly on the screen. This produces some interesting results, especially if zero page is displayed! Remember that the value put into R12,R13 is the actual memory address DIV 8.

Sideways scrolling

The whole screen can be made to move left by one character (as seen by 6845) by incrementing the screen start register. It will move one character to the right by

decrementing this register. Note that each character which moves off the left of the screen will appear on the next line up at the right of the screen. It is therefore necessary to move each of these characters down a line in software to maintain a true sideways scroll.

This scrolling technique is good for text, but may produce jumpy movements in graphics due to the limited resolution in the screen position. On a mode 0 screen, each sideways scroll moves the screen by 8 pixels. On a mode 2 screen, each 6845 character only represents 2 graphics pixels so a fairly effective hardware scroll can be used.

Mode 7 scrolling

Hardware scrolling in mode 7 is slightly more complex than in modes 0-6. To calculate the value to put into registers 12 and 13, first of all calculate the required start address in RAM (e.g. &7C28). Take the high byte and subtract &74, then EOR the result with &20. This new value should be put into R12. R13 contains the low order address byte. A similar correction factor should be applied when working out the cursor register contents.

FAST ANIMATION

Fast animation using mode 2

Mode 2 has several advantages over all of the other modes for fast animation. It is for this reason, plus the fact that all 16 colours are available that this mode is used in most fast graphics games. Provided that the programmer is prepared to put up with a 2 pixel at a time movement instead of a 1 pixel at a time movement, moving objects simplifies to moving complete bytes in memory. Consider for a moment the layout of each byte on a mode 2 screen.

	P2d	P1d	P2c	Plc	P2b	P1b	P2a	P1a
BIT	7	6	5	4	3	2	1	0

P1a-P1d are 4 bits defining the colour of pixel 1

P2a-P2d are 4 bits defining the colour of pixel 2

To move graphics sideways by one pixel involves extracting P1a-P1d from P2a-P2d. These removed bits must then be reinserted into the adjacent byte. This process is tricky and consumes a lot of processing time leading to very slow movement in all but the simplest of cases. It will be appreciated how much faster it is to simply

move a byte (2 pixels) at a time from one memory location to another, which can be done very fast indeed.

Fast animation using mode 0

Unlike mode 2, moving a byte at a time in mode 0 moves 8 pixels. Animation moving 8 pixels at a time will generally produce very uneven motion. However, since the packing of pixels in mode 0 assigns one bit per pixel, animation can be implemented by shifting all of the bits in a byte left or right by one position. This uses a 6502 'ROR' or 'ROL' instruction. The bit which moves off the edge of one byte must be put into the adjacent byte.

Wrap around

To ensure that the hardware wrap around feature operates correctly, the start of screen address must be kept within the screen boundaries. If it goes below the start of screen address then add the length of the screen to it. If it goes above the top of screen address then subtract the length of screen from it.

e.g. In mode 0, the calculated start of screen address may be &8050. Since this is outside of the screen, it should be changed to &3050 by subtracting &5000, the screen size. The amount of memory which is wrapped around is controlled by the system VIA as described in the section on the system VIA.

Hardware scroll example

The program listed below uses the hardware scroll facilities in mode 0. A line of text can be moved around the screen using the cursor keys. Note that as text moves off one side of the screen (sideways scroll), it reappears on the other side either one line up or one line down from its original position. If a true sideways scroll is required, it is necessary to move all of the bytes on the relevant side of the screen up or down one character position. During motion of the line of text in a vertical direction, there will be brief flashes of another line on the screen. This is partially due to the delay in **BASIC** between setting register 12 and register 13 on the 6845, and also because the change occurs in the middle of a screen display. The flashing will be reduced in machine code programs which wait until the frame sync period before changing any of the 6845 registers.

```
 10 REM HARDWARE SCROLL EXAMPLE IN MODE 0
 20 MODE0
 30 START=&3000
 40 PRINT"THIS TEXT CAN BE SCROLLED IN ANY DIRECTION USING THE
CURSOR KEYS"
 50 REM SET KEYS REPEAT RATE AND CURSOR KEYS TO GIVE 136 ETC.
 60 *FX4,1
 70 *FX12,3
 80 REPEAT
 90 A=INKEY(0)
100 IF A=136 THEN PROCMOVE(8)
110 IF A=137 THEN PROCMOVE (-8)
120 IF A=138 THEN PROCMOVE(-640)
130 IF A=139 THEN PROCMOVE(640)
140 UNTIL FALSE
150 DEF PROCMOVE(offset)
160 START=START+offset
170 REM IF ABOVE OFFICIAL START THEN SUBTRACT SCREEN LENGTH
181 IF START>=&8000 THEN START=STARI-&5000
190 REM IF BELOW OFFICIAL START ADDRESS, ADD SCREEN LENGTH
200 IF START<&3000 THEN START=START+&5000
190 REM MODIFY 6845 MEMORY START ADDRESS REGISTER
220 VDU23;12,START DIV 2048;0;0;0
231 VDU23;13,START MOD 2048 DIV 8;0;0;0
240 ENDPROC
```

6845 REGISTER SUMMARY TABLE

Register Number	Register name	Program unit	Data bit							
			7	6	5	4	3	2	1	0
AR	Address register	-	x	x	x	A4	A3	A2	A1	A0
R0	Horizontal total	Character	D7	D6	D5	D4	D3	D2	D1	D0
R1	Horizontal displayed	Character	D7	D6	D5	D4	D3	D2	D1	D0
R2	Horizontal sync position	Character	D7	D6	D5	D4	D3	D2	D1	D0
R3	Horizontal sync width	Character					h3	h2	h1	h0
	Vertical sync width	Scan line	v3	v2	v1	v0				
R4	Vertical total	Char. row	x	D6	D5	D4	D3	D2	D1	D0
R5	Vertical total adjust	Scan line	x	x	x	D4	D3	D2	D1	D0
R6	Vertical displayed	Char. Row	x	D6	D5	D4	D3	D2	D1	D0
R7	Vertical sync. position	Char. Row	x	D6	D5	D4	D3	D2	D1	D0
R8	Interlace mode	-							V	S
	Display enable delay	Character			d1	d0				
	Cursor enable delay	Character	c1	c0						
R9	Scan lines / character	Scan line	x	x	x	D4	D3	D2	D1	D0
R10	Cursor start	Scan line	x			s4	s3	s2	s1	s0
	Cursor blink rate	-			r					
	Cursor blink ON/OFF	-		b						
R11	Cursor end	Scan line	x	x	x	D4	D3	D2	D1	D0
R12	Screen start address H	-	x	x	h5	h4	h3	h2	h1	h0
R13	Screen start address L	-	I7	I6	I5	I4	I3	I2	I1	I0
R14	Cursor address H	-	x	x	h5	h4	h3	h2	h1	h0
R15	Cursor address L	-	I7	I6	I5	I4	I3	I2	I1	I0
R16	Light pen H	-	x	x	h5	h4	h3	h2	h1	h0
R17	Light pen L	-	I7	I6	I5	I4	I3	I2	I1	I0

x = not used

The video ULA

Sheila address &20-&21

The Video ULA is a special chip, designed by Acorn especially for use in the BBC microcomputer. It provides all of the video timing for the rest of the system (including the 6845), determines the relationship between logical and physical colours, controls the cursor width and provides Red, Green and Blue (R G B) video outputs. This section explains how the ULA is programmed for the various modes 0 - 7 and then gives an example of creating a totally new mode with 16 colours but only 10 characters across the screen. This only uses 10K of memory and therefore allows a 16 colour mode on model A micros, or large games programs on a model B which require lots of memory and full 16 colour graphics.

The video control register - SHEILA &20 write only

This 8 bit register controls which flashing colour is present at any one time, whether teletext is selected, the number of characters per line, the clock rate sent to the 6845, the width in bytes of each character and the master cursor size. `*FX154` should be used to write data into this register.

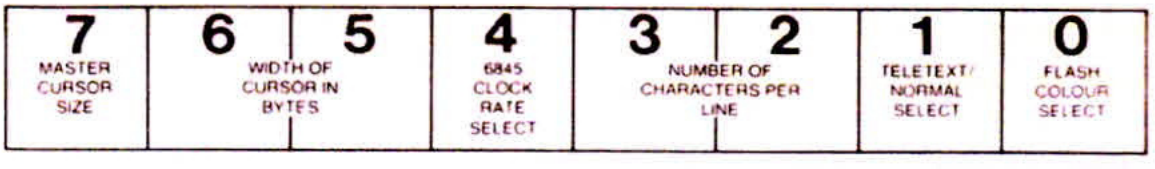


Figure 23.6 – The Video Control Register

Selected flash colour (bit 0)

This bit selects which colour of the two flashing colours is actually displayed at any particular time. It is continually changed by the operating system to generate the flashing colours. `*FX9` and `*FX10` control how long each colour is on the screen and can be defined down to one fiftieth of a second. By varying the flash rates of the colours it is possible to generate 'new' colours. This is because a flash rate of one fiftieth of a second is fast enough to fool the eye into seeing a single colour rather than two rapidly flashing colours.

0 = first colour selected

1 = second colour selected

Teletext output select (bit 1)

This bit selects whether RGB input comes from the video serialiser in the ULA or from the teletext chip.

0 = on chip serialiser used
1 = teletext input selected

Number of characters per line (bits 2,3)

These two bits determine the actual number of displayed characters per line. It is by varying this number that the new 10 character per line mode can be generated.

Bit 3	Bit 2	Number of characters per line
1	1	80
1	0	40
0	1	20
0	0	10

6845 Video Controller Chip Clock Rate Select (Bit 4)

The clock frequency sent to the 6845 can be varied using this bit.

0 = low frequency clock (modes 4-7)
1 = high frequency clock (modes 0-3)

Width of cursor in bytes (bits 5,6)

These two bits determine the number of bytes of memory required to generate a cursor width.

Bit 6	Bit 5	Number of bytes per cursor
0	0	1 (modes 0,3,4,6)
0	1	not defined
1	0	2 (modes 1,5,7)
1	1	4 (mode 2)

Master cursor width (bit 7)

If set, this bit will cause a large cursor to be generated. If reset it will cause a small cursor to be generated.

NOTE: Setting bits 5,6 and 7 to 0 will cause the cursor to vanish from the screen under ALL conditions.

General summary of the video control register

	Cursor size	Bytes per cursor		Clock speed	Number of chrs / line		TTX	Flash select	
Mode	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Hex
0	1	0	0	1	1	1	0	x	&9C
1	1	1	0	1	1	0	0	x	&D8
2	1	1	1	1	0	1	0	x	&F4
3	1	0	0	1	1	1	0	x	&9C
4	1	0	0	0	1	0	0	x	&88
5	1	1	0	0	0	1	0	x	&C4
6	1	0	0	0	1	0	0	x	&88
7	0	1	0	0	1	0	1	1	&4B

x signifies that the flash bit is changed regularly

The Palette - SHEILA &21 write only

The 'Palette' is a 64 bit RAM in the video ULA which defines the relationship between the *logical* and *actual* colours displayed on the screen. If you don't understand the difference between *logical* and *actual* colours yet, then refer to the COLOUR command in Chapter 5. `*FX155` can be used to write colour data into the Palette. It will automatically EOR the physical colour with *7* (see later). Usually, it is better to use `VDU19` or OSWORD &0C to program *logical* and *actual* colours.

The palette register consists of two 4 bit fields. Bits 0 - 3 are the *actual* colour field. Bits 4 - 7 are the *logical* colour field, as illustrated in Figure 23.7.

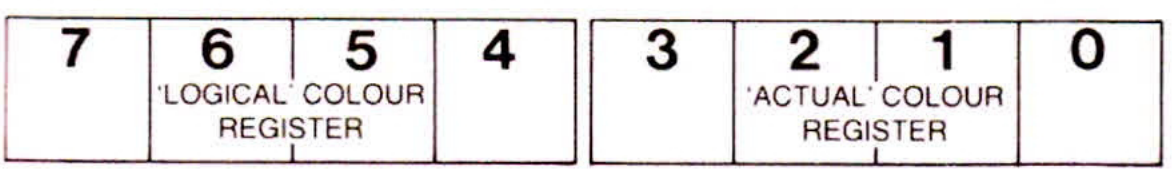


Figure 23.7 – The Palette register

Logical colour field

The following description of programming the palette only applies to direct programming using `*FX155`. If OSWORD &0C or `VDU19` are used, none of the problems which are about to be outlined will be relevant.

Programming the *logical* colour directly is easy in mode 2. The *logical* colour number then occupies the entire 4 bit field. In two colour modes 0,3,4 and 6, programming the *logical* colour directly is more complex. Bit 7 defines the *logical* colour, but bits 4,5 and 6 must be programmed to all their possible values. In other words, in order to set *logical* colour 1 to *actual* colour 5, it is necessary to program *logical* colours 9, 10, 11, 12, 13, 14 and 15 to 5. If this is not done, some parts of characters will be in one colour and other parts will be in a different colour.

Programming the *logical* colours in a four colour mode is slightly more complex. Bits 7 and 5 together contain the *logical* colour number. All other possible combinations of bits 6 and 4 must also be programmed. The following table shows how to program *logical* colours 0-3. For example, to program *logical* colour 0, it is necessary to program four separate locations in the palette.

Logical colour	bit 7	bit 6	bit 5	bit 4
0	0	0	0	0
	0	0	0	1
	0	1	0	0
	0	1	0	1
1	0	0	1	0
	0	0	1	1
	0	1	1	0
	0	1	1	1
2	1	0	0	0
	1	0	0	1
	1	1	0	0
	1	1	0	1
3	1	0	1	0
	1	0	1	1
	1	1	1	0
	1	1	1	1

General Summary for *logical* colour programming

Mode	Bit 7	Bit 6	Bit 5	Bit 4
2 colour	Logical colour Bit 0	x	x	x
4 colour	Logical colour Bit 1	x	Logical colour Bit 0	x
16 colour	Logical colour Bit 3	Logical colour Bit 2	Logical colour Bit 1	Logical colour Bit 0

Physical colour field

The *physical* colours are:

&00(0)	black
&01(1)	red
&02(2)	green
&03(3)	yellow (green—red)
&04(4)	blue
&05(5)	magenta (red—blue)
&06(6)	cyan (green—blue)
&07(7)	white
&08(8)	flashing black—white
&09(9)	flashing red—cyan
&0A(10)	flashing green—magenta
&0B(11)	flashing yellow—blue
&0C(12)	flashing blue—yellow
&0D(13)	flashing magenta—green
&0E(14)	lashing cyan—red
&0F(15)	flashing white—black

It is these colour numbers which should be used with `*FX155`. Note however that the actual number sent to the Palette is the above number EOR &07, i.e. with the three colour bits inverted and the flash bit as above.

Some interesting effects using the palette

Because of the necessity to program 4 different palette locations for each colour in a four colour mode, some 'nasty' effects can be produced on the screen if all four locations are not programmed with the same colour. To illustrate this point, try displaying four colours on the screen at once, then run this line of **BASIC**:

```
A%=155: REPEAT: X%=RND(255): CALL &FFF4: UNTIL0
```

'MODE 8' Implementation example

This example program will set up a brand new mode which has been nominated as 'MODE 8'. This is a full 16 colour mode and can be implemented on a Model A as well as a Model B since *only* 10K of RAM is used. There will only be 10 characters across the screen, but printing and plotting will operate properly. One word of warning however. Do not try to redefine the text window when this mode is in use because it will not work! All error checking on window bounds will be ineffective in this mode since the operating system does not expect mode 8 to exist.

```
 10 REM CREATE 'MODE 8'
 20 REM NOTE: NO WINDOWING ALLOWED
 30 REM
 40 REM NOTE: POKING VDU VARIABLES
 50 REM IS GENERALLY ILL ADVISED.
 60 MODE 5:REM BASIC MODE
 70 REM CONFIGURE VIDEO ULA FOR 10 COLUMN, 16 CHARACTER
 80 *FX 154,224
 90 ?&360=&F:REM COLOUR MASK
100 ?&361=1:REM PIXELS PER BYTE-1
110 ?&34F=&20:REM BYTES PER CHARACTER (4 wide x 8 high)
120 ?&363=&55:REM GRAPHICS RIGHT MASK
130 ?&362=&AA:REM GRAPHICS LEFT MASK
140 ?&30A=9:REM NO. OF CHARS PER LINE
150 VDU 20
160 REM DEMO
170 MOVE 0,0:DRAW 640,512:DRAW 1279,0
180 PRINT TAB(1,2);
190 A$="***HelloThere***"
200 COLOUR 129
210 FOR A%=1 TO 16
220 IF A%=9 THEN PRINT TAB(1,8);
230 COLOUR A%-1
240 PRINT MID$(A$,A%,1);
250 NEXT A%
260 PRINT
```

The serial system

Sheila address &08-&1F

The serial system on the BBC microcomputer deals with the transmission and reception of asynchronous serial data. Serial data is used in conjunction with the cassette and RS423 interfaces where it is impractical to use the 8 databus lines. The heart of the system is the 6850 Asynchronous Communications Interface Adapter (ACIA). This chip interfaces serial data to the 8 bit parallel processor bus. The input and output devices (i.e. cassette or RS423) and baud rates are all defined by the serial ULA. As with all the other hardware interfaces in this guide, the official operating system commands should be used to communicate with the chip registers. This will allow programs written in this way to operate over the Tube.

General description of a serial interface

This general description aims to introduce the terms which are used throughout the rest of this chapter. It is most relevant to the RS423 interface. A serial interface will usually consist of a data IN line, data OUT line and a ground connection. This is the barest minimum for a bi-directional serial link. Handshaking lines are often supplied as well. The RTS (Request to send) goes low when the BBC micro is ready to accept data on the RS423 (it acts as a data carrier enable for the cassette). The other device can control when data is sent to it via the CTS (Clear to send) line. The device pulls this low to indicate that it can receive data and pulls it high to indicate that it is unable to receive data (e.g. because its input buffer is full). Data should therefore only be sent out from the BBC microcomputer if the CTS input is low. OSBYTE &CB controls the serial handshaking and OSBYTE &CC suppresses all RS423 input.

Each character is transmitted serially according to a predefined format. A start bit indicates that a character will follow. 7 or 8 bits of character are then sent followed by a parity bit, if parity is selected. This parity bit may be set to either 1 or 0 to indicate whether the number of high bits in the character were odd or even. The parity bit (if selected) is then followed by one or two stop bits. The idea behind using a parity bit is that the receiving device can check that parity is correct. If it is incorrect then an error has occurred between the transmitter and receiver. Figure 23.8 illustrates the format of an 8 bit word with odd parity and 1 stop bit selected.

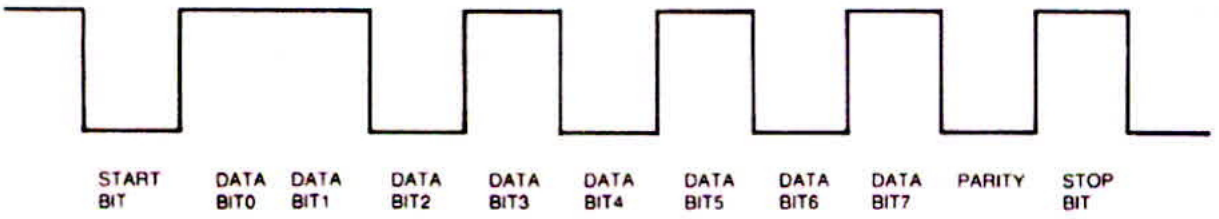


Figure 23.8 – Timing diagram for a serial character

Line termination

If very long lines are used to connect devices over an RS423 link, then it may be necessary to terminate the line. This will only be necessary in exceptional circumstances where both high baud rates and long line lengths are in use. For more information, refer to Appendix G, which describes the necessary links.

The 6850 ACIA (Asynchronous Communications Interface Adapter)

This chip performs two basic functions. It converts 8 bit parallel data to serial data, which it then transmits. It also converts a serial input stream into parallel data for the system bus, automatically providing the formatting and error checking.

Transmit data register (TDR) - Sheila &09 write only

A character may be written into the Transmit Data Register (TDR) if a status read operation has indicated that the TDR is empty. OSBYTE &97 (151) should be used to perform the write. This character is transferred to a serial shift register where it is transmitted, preceded by a start bit and followed by one or two stop bits. Internal parity (odd or even) can optionally be added to the character and will appear after the last data bit and before the first stop bit. After the first character has been written to the TDR, the Status register can be checked for a Transmit Data Register Empty condition. If the register is empty, another character can be loaded for transmission even though the first character is in the process of being transmitted. The second character will automatically be transferred into the Shift Register when the first character transmission is complete.

Receive data register (RDR) - Sheila &09 read only

Data is received from a peripheral via the cassette or RS423 serial interfaces. A divide by 64 clock ratio is provided to select the baud rate from the serial ULA, and divide by 16 or 1 ratios are provided as well. The 6850 waits until a full half of the

start bit has been received before it synchronises its clock to the bit time. As a character is being received, parity (odd or even) will be checked and any errors will be available in the Status register. When parity has been selected for an S bit word (7 bits plus parity), the 6850 sets bit 8 to 0 so that only the 7 bit data is transferred to the 6502. To read from the RDR, OSBYTE &96 (150) should be used.

The control register - Sheila &08 write only

The 8 bit 6850 control register determines the function of the transmitter, receiver, interrupts and the RS423 Request to send (RTS). Since this is a write only register, it is not possible to read its current contents directly. There is a way of determining its current contents via the operating system. OSBYTE &9C (156) is a powerful command to do this. The X and Y registers must be set, the old contents of the control register are then AND Y EOR X. Y therefore masks bits and X toggles bits. Setting Y=&FF and X=0 will generate no change at all. The routine returns with the old value of the control register in the X register. This register is normally set to &56 when using a cassette based system which isn't in the process of transmitting or receiving anything.

Counter Divide Select Bits (CR0 and CR1)

These bits determine the divide ratios used in both the transmitter and receiver sections of the ACIA. Additionally, these bits provide a master reset which initialises the transmitter, receiver and status register. The clock division rate is normally set to 64 whilst the RS423 system is in operation. The cassette system selects between 300 and 1200 baud using this division ratio. The serial ULA is always set to 300 baud for cassette, so division by 64 actually generates 300 baud. Division by 16 makes it 4 times faster so 1200 baud is generated. Division by 1 would make it a further 16 times faster, i.e. 19200 baud but the cassette will not operate at this speed.

CR1	CR0	Function
0	0	divide by 1
0	1	divide by 16
1	0	divide by 64
1	1	Master reset

Word Select Bits (CR2, CR3 and CR4)

Select bits are used to select word length, parity and the number of stop bits. Any changes become effective immediately.

CR4	CR3	CR2	Function
0	0	0	7 bits + even parity + 2 stop bits
0	0	1	7 bits + odd parity + 2 stop bits
0	1	0	7 bits + even parity + 1 stop bit
0	1	1	7 bits + odd parity + 1 stop bit
1	0	0	8 bits + 2 stop bits
1	0	1	8 bits + 1 stop bit
1	1	0	8 bits + even parity + 1 stop bit
1	1	1	8 bits + odd parity + 1 stop bit

Transmitter Control Bits (CR5 and CR6)

Two transmitter control bits provide control of the interrupt from the Transmit Data Register Empty condition, the RTS output, and the transmission of a break level (space).

CR6	CR5	Function
0	0	RTS = low, transmitting interrupt disabled
0	1	RTS = low, transmitting interrupt enabled
1	0	RTS = high, transmitting interrupt disabled
1	1	RTS = low transmits a break level on the transmit data output. Transmitting interrupt is disabled.

Receive Interrupt Enable Bit (CR7)

The conditions of receive data register full, overrun, or a low to high transition on the Data Carrier Detect (DCD) signal line are enabled by a high level bit in this position.

The status register - Sheila &08 read only

Information on the status of the 6850 is available from this register.

Receive Data Register Full (RDRF) Bit 0

The RDRF register indicates that received data has been transferred to the Receive Data Register. RDRF is cleared after the processor has read from the Receive Data Register or by a master reset. DCD being high also causes RDRF to indicate empty.

Transmit Data Register Empty (TDRE) Bit 1

This bit goes high to indicate that the Transmit Data Register contents have been transferred and that new data may now be entered. The low state indicates that the TDR is full.

Data Carrier Detect (DCD) Bit 2

When DCD goes high it indicates that the carrier is not present from the cassette input. It will always be low when the RS423 interface is selected.

Clear To Send (CTS) Bit 3

This is always low when using the cassette. On the RS423 this bit indicates that the RS423 is Clear To Send data out while this bit is low. Master reset doesn't affect the CTS bit since it is an external input.

Framing Error (FE) Bit 4

The Framing Error indicates that the received character was incorrectly framed by a start and stop bit. The error persists throughout the time that the associated character is available in the RDR.

Receiver Overrun (OVRN) Bit 5

This indicates that a character, or number of characters were received but not read from the receive data register. Character synchronisation is maintained during overrun. The overrun indication is reset after the reading of data from the Receive Data Register or after a Master reset.

Parity Error (PE) Bit 6

This bit indicates that the number of ones in the character doesn't agree with the preselected odd or even parity. The parity error is present whilst the character is in the RDR. If no parity is selected then both transmitter parity output generation and receiver parity input checks are inhibited.

Interrupt Request (IRQ) Bit 7

This indicates the state of the IRQ output. Whenever the IRQ output is low the IRQ bit is high. IRQ is cleared by a read operation to the Receive Data Register or a write operation the Transmit Data Register.

Note that OSBYTE &E8 is used to mask the 6850 ACIA IRQ.

The serial ULA - SHEILA &10

The serial ULA performs several functions related to the cassette and RS423 interfaces. It allows the input to the 6850 to be switched between cassette and RS423. It produces the transmit and receive data clocks for the 6850, thereby defining the baud rate. This ULA synthesises the data carrier signal for the cassette recording and has a data *separator* and *run in* detector when playing back cassette tapes. It produces the data carrier present signal from the cassette whenever a prerecorded program is played back.

The Serial ULA is operated from a single 8 bit control register. The various bits operate as follows:

Serial ULA bits 0-2

These define the transmit baud rate so that 000 generates 19200 baud and 111 generates 75 baud. Note that this relies upon the 6850 control register being set to divide the incoming clock signal by 64. `*FX8` is used to select the transmit baud rate on an RS423 input.

Serial ULA bits 3-5

These operate in a similar way to bits 0-2 except that they define the receiver baud rate. `*FX7` is used to select the receiving baud rate for the RS423 interface.

Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Baudrate
0	0	0	0	0	0	**19200**
1	0	0	1	0	0	**9600**
0	1	0	0	1	0	**4800**
1	1	0	1	1	0	**2400**
0	0	1	0	0	1	**1200**
1	0	1	1	0	1	**300**
0	1	1	0	1	1	**150**
1	1	1	1	1	1	**75**

Serial ULA bit 6

This selects between the cassette or RS423 system. If it is set to 0 then the cassette system is selected. If it is set to 1 the RS423 is selected. Normally, the cassette will only be selected when input from or output to cassette is in progress under the cassette filing system. OSBYTE &CD is provided to select between the RS423 and cassette serial systems.

Serial ULA bit 7

The cassette motor relay and LED can be turned on by setting this bit to 1, or off by setting it to 0. Note that the command `*MOTOR` (`*FX137` or OSBYTE &89) is available to do this.

The 'BUG' fix required when using the cassette system

On early versions of the operating system, there was a bug which led to erroneous recording of programs on cassettes. This has been corrected on later versions of the operating system (1.0 onwards) but it is necessary to know how to fix the bug if the cassette hardware is being used directly from a user program.

The problem occurred because it was possible for the serial ULA to get out of sync for a few bits when the 6850 divide bits were changed. This tended to corrupt the first character of the first block in a SAVE, or the first character of any block during sequential access (since the 6850 is reset for each block during putbytes). The cure is to write a dummy byte to tape at the start of a SAVE and the start of every block during putbytes. If the leader of a prerecorded program is played back. a run in tone followed by a blip (the dummy byte) followed by more run in tone will be heard. It is necessary to have a run in period of high tone after the dummy byte. Preferably this should be done by polling the 6850 to check if the TDR is empty, since it is difficult to accomplish if the 6850 is continually interrupting. The 6850 can then be turned on to interrupt just before starting the block write operation.

Paged ROM select register

Sheila address &30

This 4 bit write only register determines which paged ROM is switched into the memory map (e.g. **BASIC**, FORTH or LISP etc.). Up to 16 paged ROMS are therefore catered for, 4 of which are on the main circuit board. The operating system keeps track of which paged ROM is being used at any one time, so it will change the value in this register quite often. It is not advisable to POKE directly to this register, especially from **BASIC** since it is likely to crash the machine.

The ROM sockets on the main BBC microcomputer circuit board hold paged ROM numbers 12, 13, 14 and 15. These correspond to IC52, IC88, IC100 and IC101 respectively. ROM number 15 is the highest priority ROM.

There is an *official* way to read a byte from any paged ROM. This is by **CALL**ing the routine OSRDRM at &FFB9 with the relevant paged ROM number in the Y register and the address in the paged ROM in locations &F6 and &F7. The value in the byte at this location will be returned in the A register. For more details about paged ROMs, refer to Chapter 20.

The 6522 Versatile Interface Adapters

Sheila addresses &40-&7F

There are two 6522 VIAs (Versatile Interface Adapters) inside the BBC Micro. One of these is dedicated to the MOS and controls the keyboard, sound, speech, joystick fire buttons etc. The other drives the parallel printer port and the user port. The devices connected to each VIA are therefore completely different. The 6522 by itself will be considered first of all, since it applies to both units. Separate sections on the MOS VIA and the printer/user VIA then follow on.

6522 Versatile interface adapters in general

Each VIA chip is housed inside a large 40 pin package. It contains two fully programmable bi-directional 8 bit I/O ports. These are designated port A and port B, each one of which has its own 'handshaking' capability. There are two 16 bit programmable timer/counters, a serial/parallel or parallel/serial shift register and latched input/output registers.

Pin descriptions

PA0-PA7 (peripheral port A)

These 8 lines can be individually programmed as inputs or outputs under control of a Data Direction Register. The logic level on the output pins is controlled by an output register and input data can be latched into an internal register under control of the CA1 line. These various modes of operation are all controlled via internal control registers which are programmed by the 6502.

CA1, CA2 (port A control lines)

These two lines can act either as interrupt inputs or as handshake outputs. Each line controls an internal interrupt flag with a corresponding interrupt enable bit. In addition, CA1 controls the latching of data on port A input lines.

PB0-PB7 (peripheral port B)

The 8 bi-directional port B lines are controlled by an output register and a data direction register in a similar way to port A. The logic level of the PB7 output signal can also be controlled by one of the interval timers. The second timer can be programmed to count pulses on the PB6 input. These outputs are capable of sourcing up to 1 mA at 1.5 volts in the output mode. This allows direct drive of

Darlington transistor circuits. Note that only the port B lines can provide 1mA, the port A lines cannot.

CB1, CB2 (port B control lines)

The port B control lines act as interrupt inputs or as handshake outputs just like port A. They can also be programmed to act as a serial port under the control of the shift register. These lines cannot source 1mA either.

Electrical specification

Inputs

Input voltage for logic 1	= 2.4 VDC minimum
Input voltage for logic 0	= 0.4 VDC maximum
Maximum required input current	= 1.8 mA

Outputs

Output logic 1 voltage	= 2.4 VDC minimum at a load of 100 uA maximum (except PB0-PB7)
Output logic 0 voltage	= 0.4 VDC maximum when sinking 1.6 mA
Current sinking capability	= 1.6 mA minimum

Functional description

Reg. Num	RS Coding				Reg. Desig.	Description	
	RS3	RS2	RS1	RS0		Write	Read
0	0	0	0	0	ORB/IRB	Output Register "B"	Input Register "B"
1	0	0	0	1	ORA/IRA	Output Register "A"	Input Register "A"
2	0	0	1	0	DDRB	Direction Register "B"	
3	0	0	1	1	DDRA	Direction Register "A"	
4	0	1	0	0	T1C-L	T1 Low-Order Latches	T1 Low-Order Counter
5	0	1	0	1	T1C-H	T1High-Order Counter	
6	0	1	1	0	T1L-L	T1 Low-Order Latches	
7	0	1	1	1	T1L-H	T1 High-Order Latches	
8	1	0	0	0	T2C-L	T2 Low-Order Latches	T2 Low-Order Counter
9	1	0	0	1	T2C-H	T2 High-Order Counter	
10	1	0	1	0	SR	Shift Register	
11	1	0	1	1	ACR	Auxiliary Control Register	
12	1	1	0	0	PCR	Peripheral Control Register	
13	1	1	0	1	IFR	Interrupt Flag Register	
14	1	1	1	0	IER	Interrupt Enable Register	
15	1	1	1	1	ORA/IRA	Same as Reg 1 Except No Handshake	

Figure 23.9 - 6522 Internal Register Summary

Operation of port A and port B

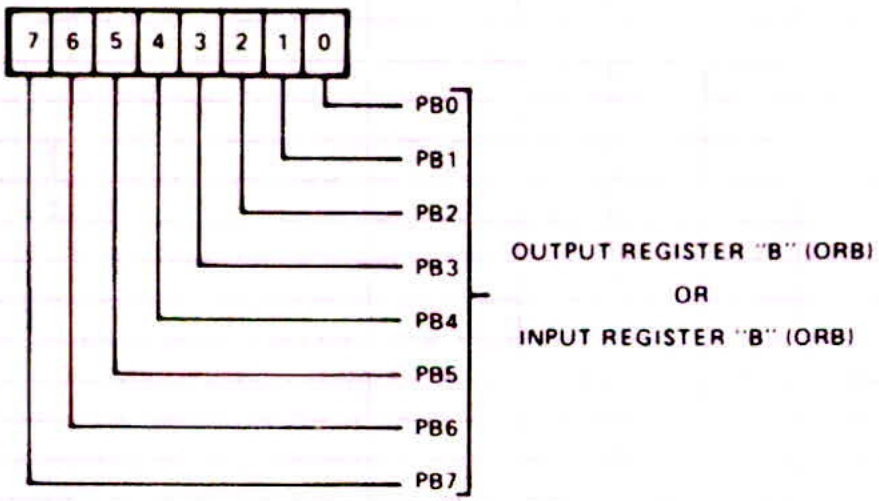
There are two data direction registers DDRA and DDRB which specify whether the peripheral pins are to operate as inputs or outputs. Placing a '0' in a bit of a DDR will cause the corresponding bit of that port to be defined as an input. A '1' will cause it to be defined as an output.

Each of the port's I/O pins is controlled by a bit in an output register (ORA or ORB) and an input register (IRA or IRB). When programmed as an output, a port line will be controlled by the corresponding bit in the output register. If the line is defined as an input then writing data into its output register will have no effect. Reading from a peripheral port will read the value of the input register (IRA or IRB). With input latching disabled IRA will contain the value present at PA0-PA7 when the read is performed. If input latching is enabled then IRA will contain the value present at PA0-PA7 when the latching occurred (via CA1).

The IRB register is similar to the IRA register, but there is a difference for pins programmed as outputs. When reading IRA, it is *the voltage level* on PA0-PA7 which determines the level read back. When reading IRB, it is always the bit in the output register which is read back. This means that with loads which pull an output

'1' low or an output '0' high. reading IRA may indicate a different logic level to that written to the output. Reading IRB will however always read back the value programmed no matter what loading is applied to the pin.

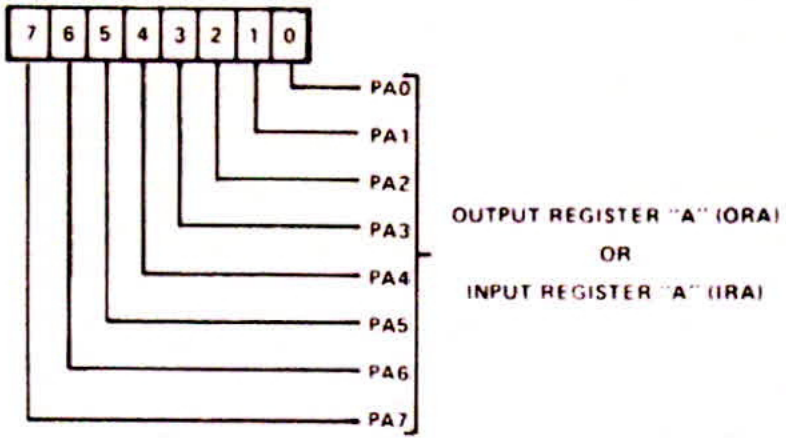
REG 0 – ORB/IRB



Pin Data Direction Selection	WRITE	READ
DDRB = "1" (OUTPUT)	MPU writes Output Level (ORB)	MPU reads output register bit in ORB. Pin level has no affect.
DDRB = "0" (INPUT) (Input latching disabled)	MPU writes into ORB, but no effect on pin level, until DDRB changed	MPU reads input level on PB pin
DDRB = "0" (INPUT) (Input latching enabled)		MPU reads IRB bit, which is the level of the PB pin at the time of the last CB1 active transition

Figure 23.10 – Port B Input/Output

REG 1 – ORA/IRA



Pin Data Direction Selection	WRITE	READ
DDRA = "1" (OUTPUT) (Input latching disabled)	MPU writes Output Level (ORA)	MPU reads level on PA pin
DDRA = "1" (OUTPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition
DDRA = "0" (INPUT) (Input latching disabled)	MPU writes into ORA, but no effect on pin level, until DDRA changed	MPU reads level on PA pin
DDRA = "0" (INPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition

Figure 23.11 – Port A Input/Output

REG 2 (DDRB) AND REG 3 (DDRA)

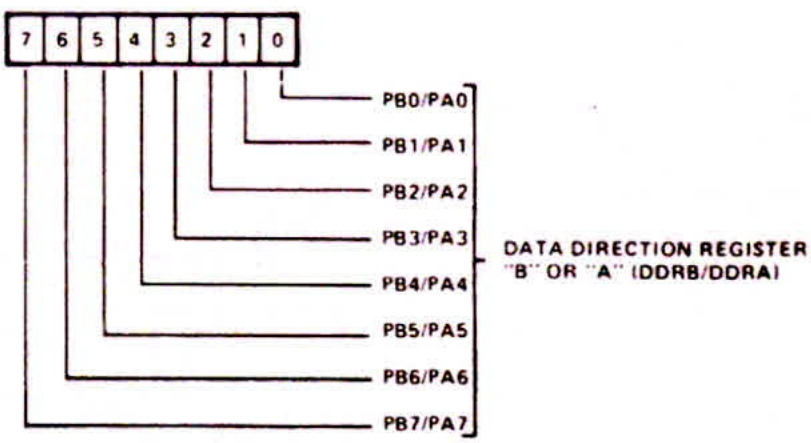


Figure 23.12 – Data Direction Registers

Write handshaking data transfer

Handshaking allows data transfers between two asynchronous devices. Write handshaking operates with 'data ready' and 'data taken' signals. The 6522 provides the 'data ready' (CA2 or CB2) signal and accepts the 'data taken' (CA1 or CB1) signal from the peripheral device. This 'data taken' signal sets the interrupt flag and clears the 'data ready' output. See the timing diagram figure 23.13.

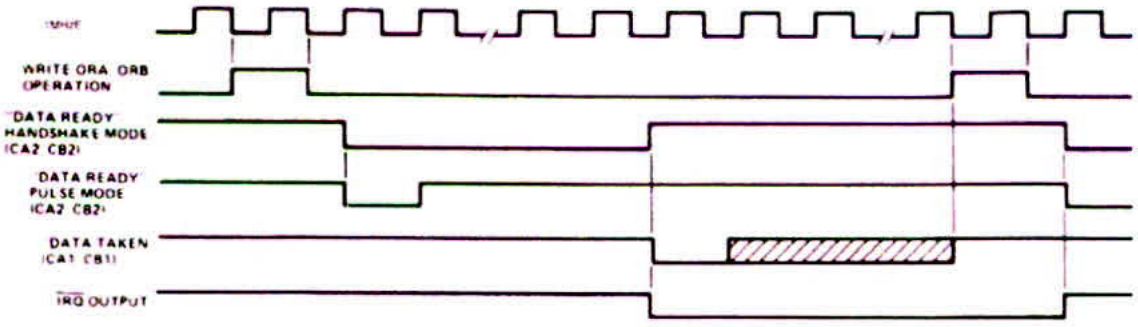


Figure 23.13 – Write Handshake Timing

Selection of operating modes for CA1, CA2, CB1 and CB2 is controlled by the Peripheral Control Register, see figure 23.14.

REG 12 – PERIPHERAL CONTROL REGISTER

7	6	5	4	3	2	1	0

CB2 CONTROL

7	6	5	OPERATION
0	0	0	INPUT-NEGATIVE ACTIVE EDGE
0	0	1	INDEPENDENT INTERRUPT INPUT-NEG EDGE
0	1	0	INPUT-POSITIVE ACTIVE EDGE
0	1	1	INDEPENDENT INTERRUPT INPUT-POS EDGE
1	0	0	HANDSHAKE OUTPUT
1	0	1	PULSE OUTPUT
1	1	0	LOW OUTPUT
1	1	1	HIGH OUTPUT

CB1 INTERRUPT CONTROL

0 = NEGATIVE ACTIVE EDGE
1 = POSITIVE ACTIVE EDGE

CA1 INTERRUPT CONTROL

0 = NEGATIVE ACTIVE EDGE
1 = POSITIVE ACTIVE EDGE

CA2 CONTROL

3	2	1	OPERATION
0	0	0	INPUT NEGATIVE ACTIVE EDGE
0	0	1	INDEPENDENT INTERRUPT INPUT-NEG EDGE
0	1	0	INPUT POSITIVE ACTIVE EDGE
0	1	1	INDEPENDENT INTERRUPT INPUT POS EDGE
1	0	0	HANDSHAKE OUTPUT
1	0	1	PULSE OUTPUT
1	1	0	LOW OUTPUT
1	1	1	HIGH OUTPUT

Figure 23.14 – CA1, CA2, CB1, CB2 Control

Timer operation

The interval timer, referred to from now on as 'T1', consists of two 8 bit latches and a 16 bit counter. After it has been loaded, the counter decrements at the system clock rate (1 MHz) until it reaches zero. When it reaches zero, an interrupt flag will be set and an interrupt will be requested of the 6502, if enabled. The timer then disables any further interrupts, or automatically transfers the contents of the latches into the counter and continues to decrement. The timer may also be programmed to invert the output level on an output line every time its count reaches zero. Figure 23.15 and figure 23.16 illustrate the T1 counter and latches.

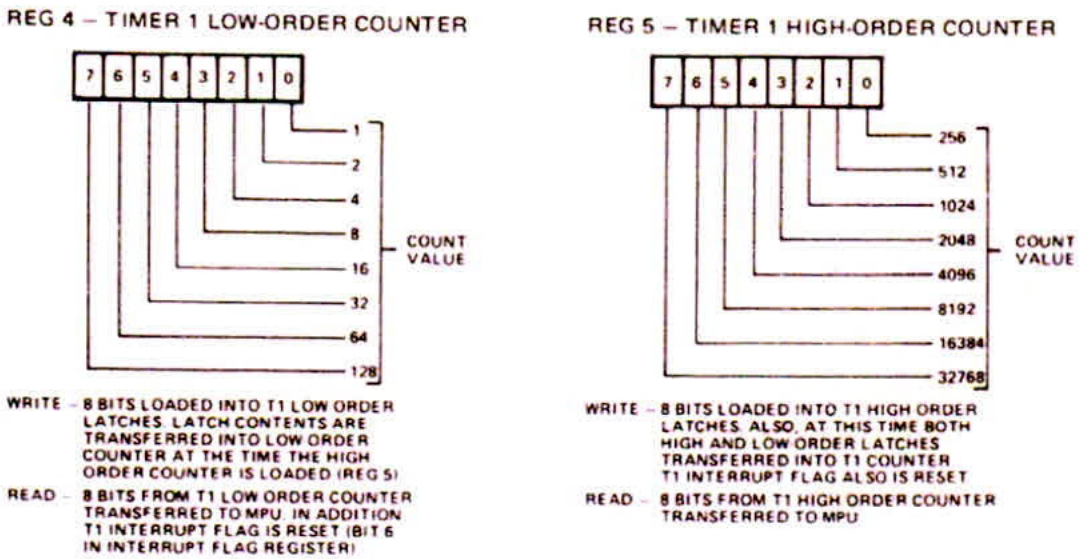


Figure 23.15 – T1 Counter Registers

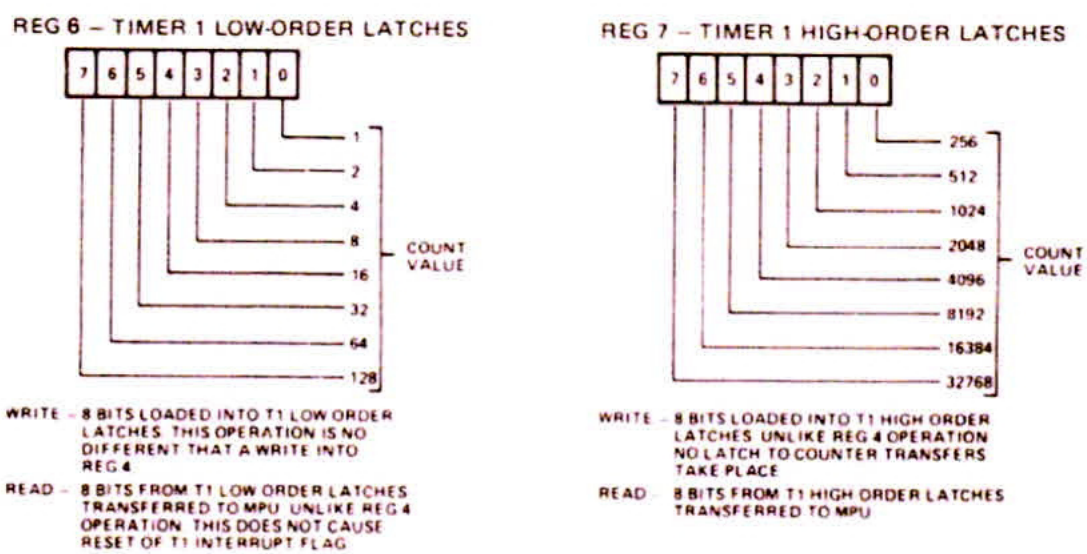


Figure 23.16 – T1 Latch Registers

Timer 1 one-shot mode

This mode allows a single interrupt to be generated for each timer load operation. The delay between writing T1C-H and generation of the interrupt to the 6502 is a direct function of the data loaded into the counter. T1 can be programmed to produce a single negative pulse on the PB7 peripheral pin as well as generating a single interrupt. With output enabled (ACR=1), writing T1C-H will cause PB7 to go low. PB7 will go high again when T1 'times out'. The overall result of this is a programmable width pulse on PB7.

Writing into the high order latch has no effect on the operation of T1 in the one-shot mode. It is however necessary to ensure that the low order latch contains the correct data before initiating the countdown by writing T1C-H. When the 6502 writes into the high order counter, the T1 interrupt flag is cleared, the contents of the low order latch are transferred into the low order counter, and the timer begins to decrement at 1MHz. If PB7 output is enabled then it will go low after the write operation. Upon reaching zero, the T1 interrupt flag is set, an interrupt is generated (if enabled) and PB7 goes high. The counter continues to decrement at the system clock rate. The 6502 is then able to read the contents of the counter to determine the time since the interrupt occurred. The T1 interrupt must be cleared before it can be set again.

Timer 1 free-run mode

The advantage of having latches which *remember* the initial value put into the counter is that the initial value can be restored after the counter has decremented to zero. If this is done automatically then the timer enters a free-running mode. In the free-running mode, PB7 is inverted and the interrupt flag is set each time the counter has decremented to zero. The contents of the 16 bit latch are then transferred to the counter, which decrements to zero again and so on. This produces a true square wave of variable frequency on the PB7 output. The interrupt flag can be cleared by writing T1C-H. by reading T1C-L, or by writing directly into the flag as will be described.

All of the timers in the 6522 can be retriggered. This means that rewriting the value in the counter will always re-initialise the time-out period. Time-out will therefore be completely inhibited if the processor continues to rewrite the timer before it reaches zero. T1 operates in this way if the 6502 writes into the high order counter (T1C-H). If the 6502 only loads the latches, this will not affect the counter until the

next time zero is reached. The timer can be read without affecting its value. This can be very useful because the new timer time doesn't come into effect until zero is reached. If the 6502 responds to each interrupt by programming a new value into the latches, the period of the next half cycle on the PB7 output will be determined. Waveforms with complex mark - space ratios can be generated in this way.

Timer 2 operation

Timer 2 operates either as an interval timer (in the one-shot mode only) or as a counter for counting negative pulses on the PB6 pin. A single control bit in the Auxiliary Control Register selects between these two modes. Timer 2 comprises a 'write only' low order latch (T2L-L), a 'read only' low order counter and a read/write high order counter. The counter register contents are decremented at 1 MHz. Figure 23.17 illustrates the timer 2 counter registers.

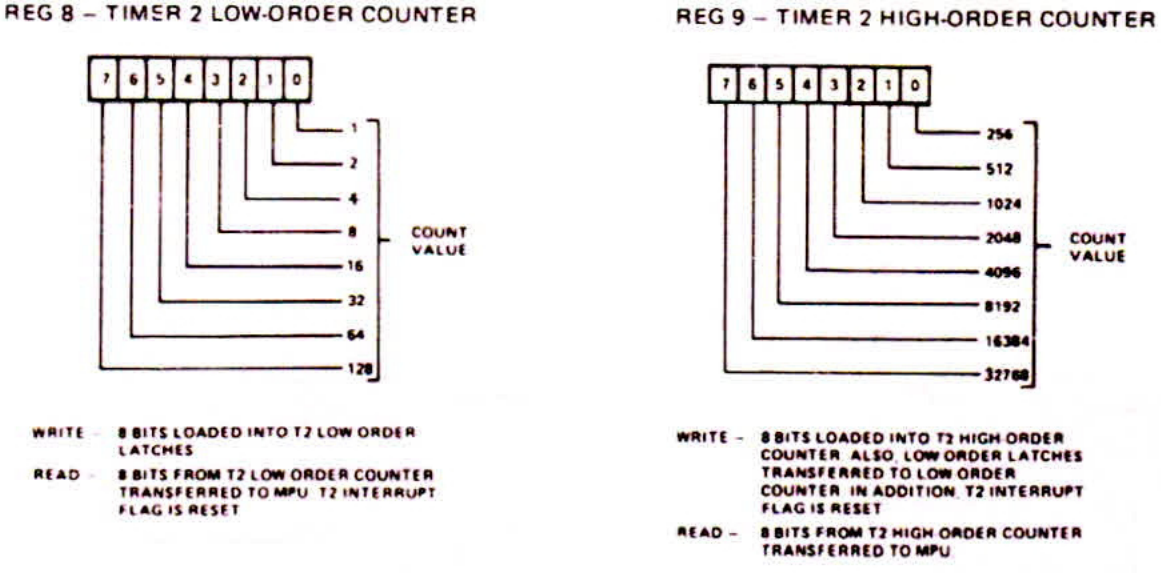


Figure 23.17 – T2 Counter Registers

Timer 2 one-shot mode

In the one-shot mode, the operation of timer 2 is similar to that of timer 1. T2 provides a single interrupt for each time out after T2C-H had been set. The counter continues to decrement after time-out, but the interrupt is disabled after the initial time-out so that it will not be set again each time that the timer decrements through zero. T2C-H must be rewritten to re-enable the interrupt flag. The interrupt flag is cleared by reading T2C-L or by writing T2C-H.

Timer 2 pulse counting mode

In this mode, T2 counts a predetermined number of negative going pulses applied to PB6. This can be accomplished by first of all loading a number into T2. Writing into T2C-H will clear the interrupt flag and allow the counter to decrement every time that a pulse is applied to PB6. The interrupt flag is set when T2 counts down past zero. The timer continues to decrement with each pulse applied to PB6. T2C-H must be rewritten to allow the interrupt flag to set on subsequent down counts.

Shift register operation

The shift register (SR) enables serial data to be transferred into and out of the CB2 pin under the control of an internal modulo-8 counter. Pulses from an external source can be applied to CB1 to shift a bit into or out of CB2. Alternatively, with proper mode selection, shift pulses generated internally will appear on the CB1 pin for controlling external devices.

The control bits which select the various shift register operating modes are located in the Auxiliary Control Register. The configuration of the SR data bits and the SR control bits of the ACR are illustrated in Figure 23.18 and Figure 23.19.

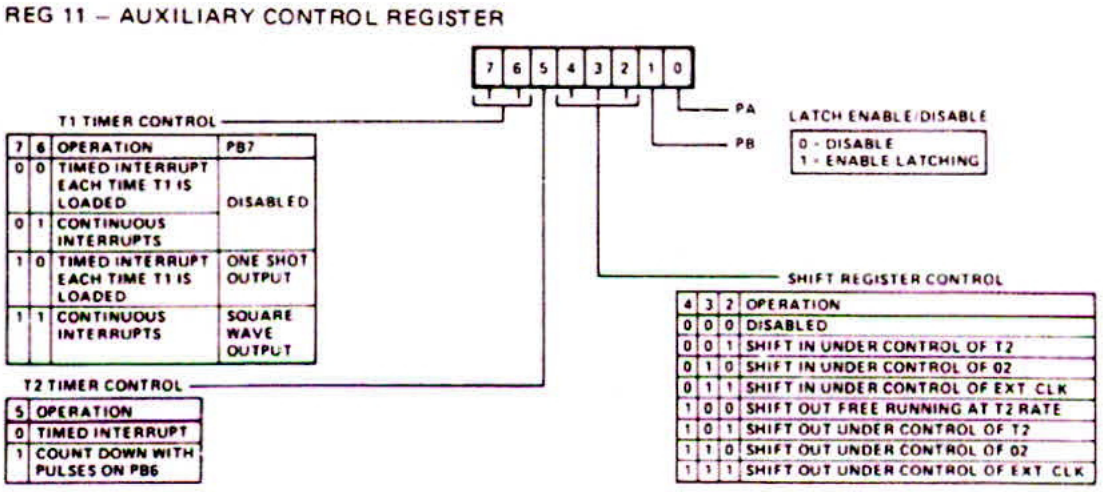


Figure 23.18 – Auxiliary Control Register

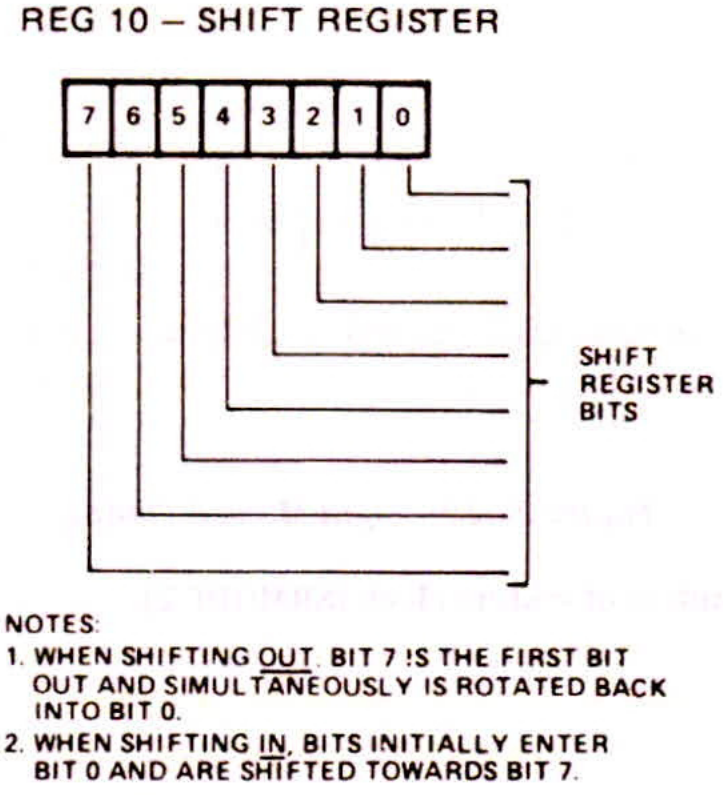


Figure 23.19 – Shift Register Control Bits

Shift register modes of operation

Shift Register Disabled (SRMODE 0)

In this mode the SR is disabled. The 6502 can however write or read the SR and the SR will shift one bit left on each CB1 positive edge. The logic level present on CB2 is shifted into bit 0. The SR interrupt flag is always disabled in this mode.

Shift in under control of T2 (SRMODE 1)

In mode 1 the shifting rate is controlled by the 8 low order bits of T2. Shift pulses are generated on the CB1 pin to control shifting in external devices. The time between transitions of this output clock is controlled by the low order T2 latch.

Reading from or writing to the SR will trigger a shifting operation if the SR flag in the IFR is set. If it isn't set then the first shift will occur when T2 next times out after a read or write SR. Data is shifted first into the low order bit of the SR, then into the next higher order bit and so on the negative edge of each shift clock pulse. The input data should then change before the next positive going edge of CB1. Data is shifted into the shift register on the positive going edge of the CB1 pulse.

After 8 CB1 clock pulses, the shift register interrupt flag will be set and an interrupt will be requested of the 6502.

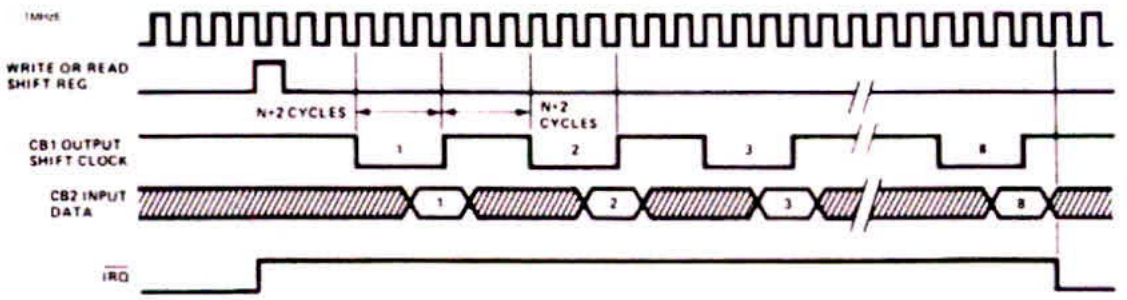


Figure 23.20 – Input Mode 2 Timing

Shift in under control of system clock (SRMODE 2)

In mode 2 the shift rate is a direct function of the 1MHz system clock. Pulses for controlling external devices are generated on the CB1 output. Timer 2 has no effect on the SR and acts as an independent interval timer. The shifting operation is triggered by reading or writing the SR. Data is first shifted into bit 0 and then into successively higher order bits on the trailing edges of system clock pulses. After 8 clock pulses, the shift register interrupt flag will be set and output clock pulses from CB1 will cease.

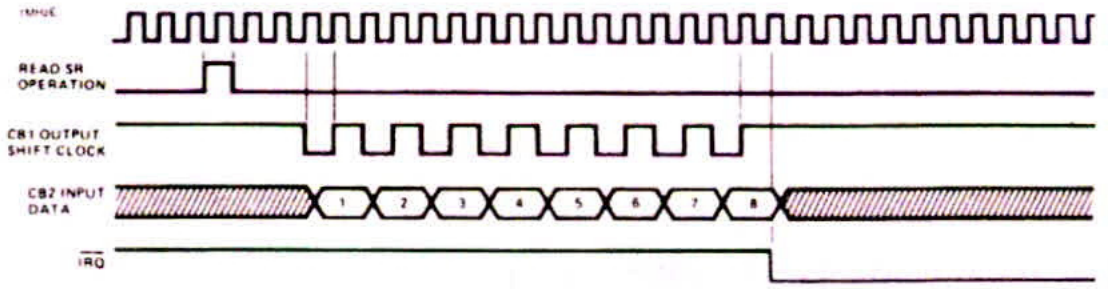


Figure 23.21 – Output Mode 2 Timing

Shift in under control of external CB1 clock (SRMODE 3)

CB1 is a clock input in mode 3 so that external devices can load the shift register at their own pace. The shift register counter will generate an interrupt each time that 8 bits have been shifted in. The SR counter does NOT stop the shifting operation, it simply operates as a pulse counter. Reading from or writing to the shift register resets the interrupt flag and initialises the SR counter to count another 8 pulses. Note that data is shifted in on the first system clock cycle following the positive

going edge of the CB1 shift pulse. Data must therefore be held stable during the first full system clock cycle after CB1 has gone high.

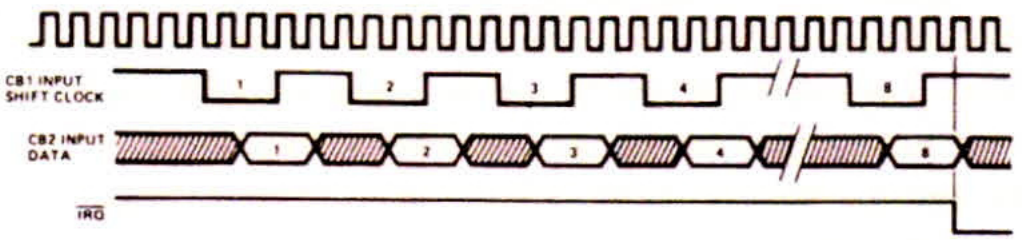


Figure 23.22 – Output Mode 3 Timing

Shift out free running at T2 clock rate (SRMODE 4)

In this mode the shift rate is controlled by timer 2 (T2). Unlike mode 5, the SR counter will not stop the shifting operation. Shift register bit 7 is recirculated back into bit 0, so the 8 bits loaded into the shift register will be clocked onto CB2 repetitively. The shift register counter is disabled in this mode.

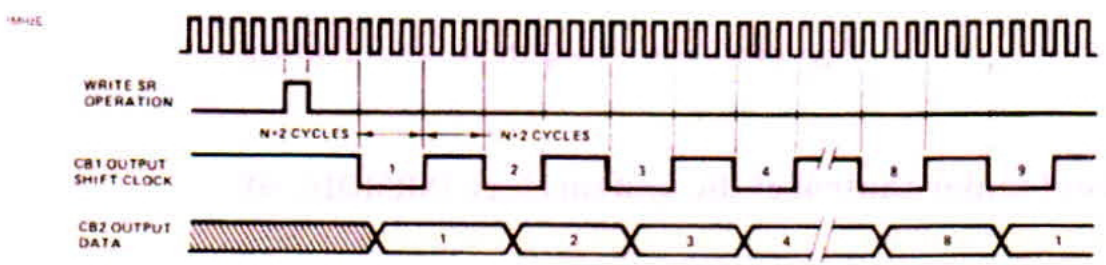


Figure 23.23 – Output Mode 4 Timing

Shift out under control of T2 (SRMODE 5)

The shift rate is controlled by T2 as in mode 4. if the SR flag in the IFR is set, then the shifting operation is triggered by the read or write of the SR. Alternatively the first shift will occur at the next timeout of T2 after a read or write of the SR. With each write or read of the SR. the SR counter is reset and 8 bits are shifted onto CB2. Eight shift pulses appear on the CB1 output to facilitate the control of shifting into external devices. When the 8 shift pulses have occurred, shifting is disabled, the SR interrupt flag is set and CB2 remains fixed at the last data bit level.

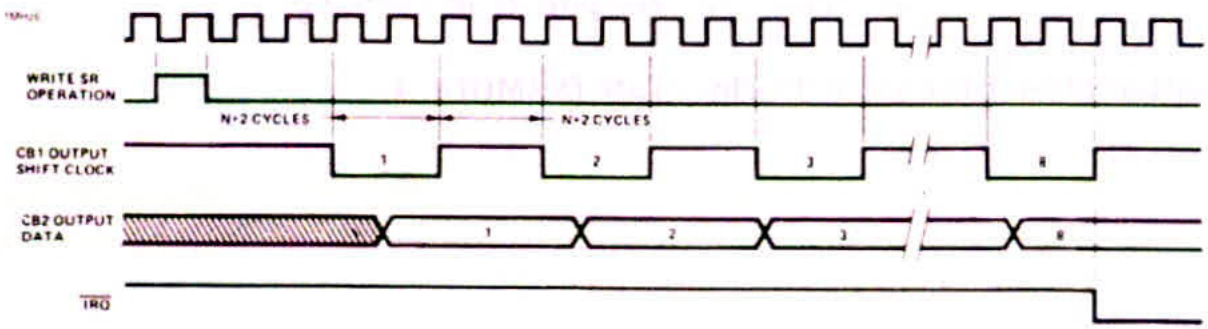


Figure 23.24 – Output Mode 5 Timing

Shift out under control of the system clock (SRMODE 6)

In this mode, the shift rate is controlled directly by the 1MHz system clock.

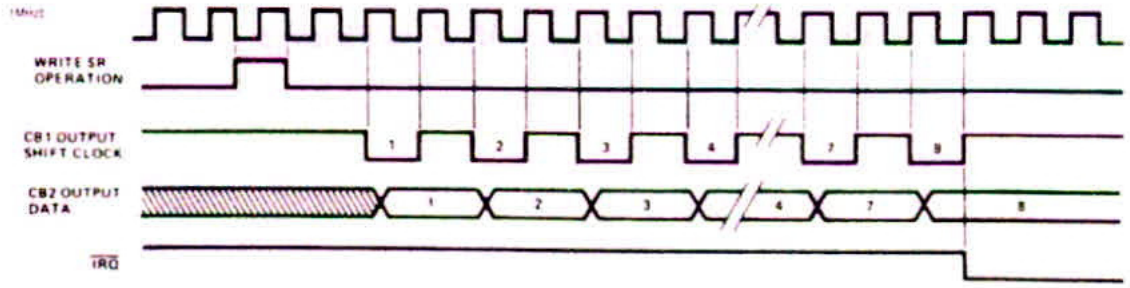


Figure 23.25 – Output Mode 6 Timing

Shift out under control of external CB1 clock (SRMODE 7)

Shifting is controlled by pulses applied to the CB1 pin by an external device in this mode. The SR interrupt flag is set each time that the SR counter counts 8 pulses, but the shifting function is not disabled. The SR interrupt flag is reset and the SR counter is initialised to begin counting the next 8 shift pulses on CB1, each time that the 6502 writes or reads the shift register. The interrupt flag is set after 8 shift pulses. The 6502 can then load the next byte of data into the shift register.

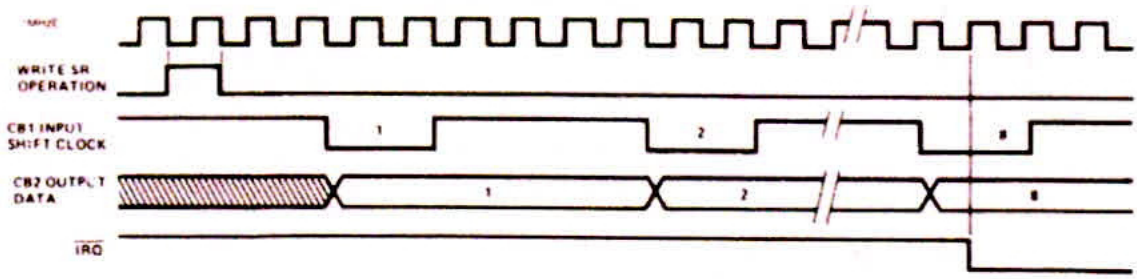


Figure 23.26 – Output Mode 7 Timing

Interrupt operation

Interrupt flags are set either by an interrupt condition in the chip (e.g. from a counter), or an interrupt condition on an input to the chip. Interrupt flags normally remain in the set condition until the interrupt has been serviced. The source of an interrupt can be determined by reading these interrupt flags in order from highest priority to lowest priority. This is best performed by reading the flag register into the processor accumulator, shifting either right or left and using conditional branch instructions to detect an active interrupt.

There is an interrupt enable bit associated with each interrupt flag. If this enable bit is set to a logic 1 and the associated interrupt occurs, then the 6502 will be interrupted. If the enable bit is set to 0 then the 6502 will not be interrupted.

All interrupt flags are contained in the interrupt flag register (IFR - see Figure 23.27). To enable the 6502 to check the 6522 without checking each bit in the IFR, bit 7 will be set to a logic 1 if the 6522 has generated the interrupt. In addition to reading the IFR, individual bits may be cleared by writing a 1 into the appropriate bit of the IFR. Note however that IFR bit *7* is not a flag as such and will not be cleared by writing a 1 into it. It can only be cleared by clearing all the flags in the register or by disabling ALL of the active interrupts.

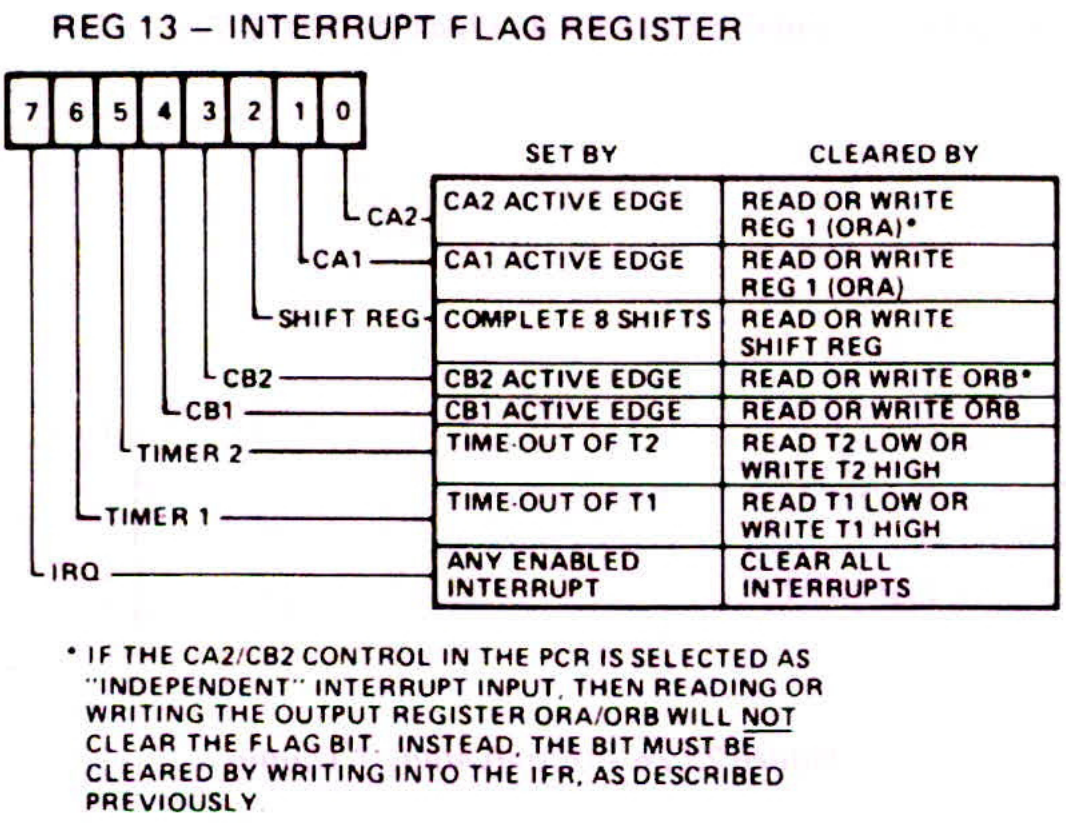
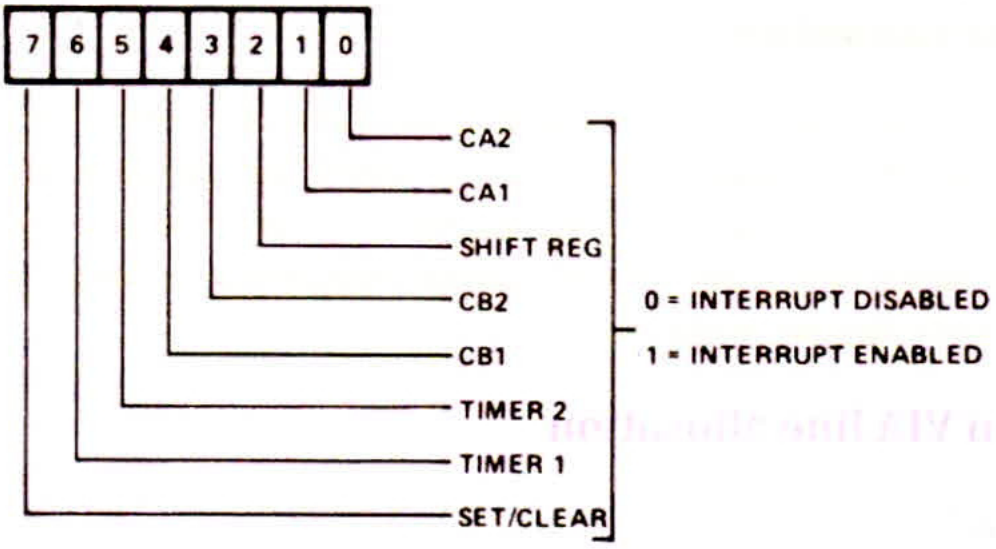


Figure 23.27 – Interrupt Flag Register

The 6502 can set or clear selected bits in the interrupt enable register without affecting the other bits. This is accomplished by writing to the IER. If bit 7 of the byte written is a 0 then each 1 in bits 0—6 will clear the corresponding bit in the IER. For each zero in bits 0-6, the corresponding bit will not be affected. Selected bits can be SET in a similar manner. In this case, bit 7 of the written byte should be set to 1. Each 1 in bits 0—6 will then SET the selected bit. A zero will cause the corresponding bit to remain unaffected. The contents of the IER can be read by the 6502. Bit 7 is then *always* read as a logic 1.

REG 14 – INTERRUPT ENABLE REGISTER



NOTES:
1. IF BIT 7 IS A "0", THEN EACH "1" IN BITS 0 - 6 DISABLES THE CORRESPONDING INTERRUPT.
2. IF BIT 7 IS A "1", THEN EACH "1" IN BITS 0 - 6 ENABLES THE CORRESPONDING INTERRUPT.
3. IF A READ OF THIS REGISTER IS DONE, BIT 7 WILL BE "1" AND ALL OTHER BITS WILL REFLECT THEIR ENABLE/DISABLE STATE.

Figure 23.28 – Interrupt Enable Register

The System VIA

Sheila addresses &40-&4F

The System VIA is responsible for a large amount of control within the BBC Micro itself. It controls the speech system, sound system and keyboard. Also, several other sections can be partially controlled from this VIA. These are the hardware scrolling, vertical sync pulse interrupt, joysticks input, end of conversion input from the ADC and a light pen strobe input.

System VIA line allocation

PA0-PA7

The 6502 CPU does not talk to the speech system, sound generator or keyboard directly over its data bus. Instead, it writes to and reads from the 8 bit port A I/O lines. This forms a 'slow' databus over which the CPU can communicate. To write to this databus, the data direction register A at Sheila &43 should set all lines as outputs. The 6502 can then write directly into output register A at Sheila &41. To read from the slow data bus, DDRA must set all lines as inputs by writing &00 to Sheila address &43. A direct read from input register A at Sheila &41 can then be made.

NOTE: Any reading or writing over this slow databus will have to be done from machine code with ALL 6502 interrupts disabled. This is because the interrupt routines themselves will make extensive use of the system VIA and keep changing the register values.

CA1 input

This is the vertical sync input from the 6845. CA1 is set up to interrupt the 6502 every 20 ms (50 Hz) as a vertical sync from the video circuitry is detected. The operating system changes the flash colours on the display in this interrupt time so that they maintain synchronisation with the rest of the picture.

CA2 input

This input comes from the keyboard circuit, and is used to generate an interrupt whenever a key is pressed. See the keyboard circuit diagram in Appendix H for more details.

PB0-PB2 outputs

These 3 outputs form the address to an 8 bit addressable latch, IC32 on the main circuit diagram. See the following 'Addressable Latch' section.

PB3 output

This output holds the data to be written to the selected addressable latch bit.

PB4 and PB5 inputs

These are the inputs from the joystick FIRE buttons. They are normally at logic 1 with no button pressed and change to 0 when a button is pressed. OSBYTE &80 can be used to read the status of the joystick fire buttons.

PB6 and PB7 inputs from the speech processor

PB6 is the speech processor 'ready' output and PB7 is from the speech processor 'interrupt' output.

CB1 input

The CB1 input is the end of conversion (EOC) signal from the 7002 analogue to digital converter. It can be used to interrupt the 6502 whenever a conversion is complete. See the section on the Analogue to Digital Converter.

CB2 input

This is the light pen strobe signal (LPSTB) from the light pen. It also connects to the 6845 video processor. CB2 can be programmed to interrupt the processor whenever a light pen strobe occurs.

The addressable latch

This 8 bit addressable latch is operated from port B lines 0-3 inclusive. PB0-PB2 are set to the required address of the output bit to be set. PB3 is set to the value which should be programmed at that bit. An example illustrating how to use this latch from **BASIC** is described in conjunction with the sound generator. The functions of the 8 output bits from this latch are:

B0 - Write Enable to the sound generator IC.
B1 - READ select on the speech processor
B2 - WRITE select on the speech processor
B3 - Keyboard write enable (see Appendix H)

B4, B5 - these two outputs define the number to be added to the start of screen address in hardware to control hardware scrolling:

Mode	Size	Start of screen	Number to add	B5	B4
0,1 and 2	20K	&3000	12K	1	1
3	16K	&4000	16K	0	0
4 and 5	10K	&5800 (or &1800)	22K	1	0
6	8K	&6000 (or &2000)	24K	0	1

B6 - Operates the CAPS lock LED
B7 - Operates the SHIFT lock LED

The 76489 sound chip

The sound chip on the BBC microcomputer is in itself a very simple chip. There are three channels for which the frequency and volume of output can be defined. There is also a fourth white noise generator. The output from all of these channels is automatically mixed on chip. The complex sound commands available from **BASIC** are very powerful but require a large amount of time to process, especially if complex envelopes are defined. In fast machine code programs it may sometimes be advantageous to write directly to the sound chip. The example program shows how this can be done. The data to be written into the sound chip is first of all put onto the slow databus. Note that interrupts are disabled before this is started. The sound generator write enable line is then pulled low for at least 8uS then pulled high again.

Tone generators

There are 3 tone generators. The frequency of each channel is determined by 10 bits of data. F9 is the most significant bit. The frequency of each channel can be calculated as:

frequency = 4000000/32 x 10 bit binary number

The volume level for each channel is variable to 16 different levels these are:

Bit A3	Bit A2	Bit A1	Bit A0	VOLUME
0	0	0	0	**15 (MAX)**
0	0	0	1	**14**
0	0	1	0	**13**
0	0	1	1	**12**
0	1	0	0	**11**
0	1	0	1	**10**
0	1	1	0	**9**
0	1	1	1	**8**
1	0	0	0	**7**
1	0	0	1	**6**
1	0	1	0	**5**
1	0	1	1	**4**
1	1	0	0	**3**
1	1	0	1	**2**
1	1	1	0	**1**
1	1	1	1	**0 (OFF)**

Noise generator

The noise generator comprises a noise source and volume control. The noise generator parameters are defined by three bits.

FB - this bit when set to '0' causes PERIODIC NOISE to be generated. When set to '1' it causes WHITE NOISE to be generated.

Noise frequency control - the noise base frequency can be defined in 4 possible states by bits NF1 and NF0.

NF1	NF0	FREQUENCY
0	0	low
0	1	medium
1	0	high
1	1	tone generator 1 frequency

Sound chip register address field

R2	R1	R0	Description
0	0	0	**Tone 3 frequency**
0	0	1	**Tone 3 volume**
0	1	0	**Tone 2 frequency**
0	1	1	**Tone 2 volume**
1	0	0	**Tone 1 frequency**
1	0	1	**Tone 1 volume**
1	1	0	**Noise control**
1	1	1	**Noise volume**

Programming byte formats

The sound generator is programmed by sending it bytes in the following format:

Frequency (First byte)

	Register Address				Data			
Bit	7	6	5	4	3	2	1	0
	1	R2	R1	R0	F3	F2	F1	F0

Frequency (Second byte)

	Data							
Bit	7	6	5	4	3	2	1	0
	0	X	F9	F8	F7	F6	F5	F4

NOTE: The second low order frequency byte may be continually updated without rewriting the first byte.

Noise source byte

	Register Address							
Bit	7	6	5	4	3	2	1	0
	1	R2	R1	R0	X	FB	NF1	NF0

Update volume level

	Register Address				Data			
Bit	7	6	5	4	3	2	1	0
	1	R2	R1	R0	A3	A2	A1	A0

Example program for direct control of the sound generator

```
 10 REM Demonstration of direct poke to Sound chip
 20 PROCINIT
 30 REPEAT
 40 INPUT"Byte to send to sound chip";A$
 50 A% = EVAL(A$)
 60 CALL DIRECT
 70 UNTIL FALSE
 80 DEF PROCINIT
 90 DIM Q% 40
100 OSBYTE = &FFF4
110 FOR C=0 TO 3 STEP 3
120 P% = Q%
130 [OPT C
140 .DIRECT SEI         \ Disable interrupts
150 PHA
160 LDA #&97
170 LDX #&43            \ Data direction register A
180 LDY #&FF            \ Set all B bits as output
190 JSR OSBYTE          \ Write to SHEILA OSBYTE CALL
200 LDX #&41            \ Output register A
210 PLA
220 TAY                 \ Y holds byte to sound chip
230 LDA #&97            \ Write to SHETLA OSBYTE CALL
240 JSR OSBYTE          \ Output to slow data bus
250 LDX #&40            \ Output register B
260 LDY #&00            \ Set sound chip write pin low
270 JSR OSBYTE
280 LDY #&08            \ Set sound chip write pin high
290 JSR OSBYTE
300 CLI                 \ Enable interrupts
310 RTS: ]
320 NEXT
330 ENDPROC
```

Run the example program and enter &80, &20 and &90 to generate a frequency at maximum volume on channel 3.

The User/Printer VIA

Sheila addresses &60-&6F

Full programming details for the 6522 VIA are described earlier in this Chapter. This brief section is designed to help anyone who specifically wishes to use the USER VIA.

PORT A - The printer port

All of the port A lines PA0-PA7 are buffered before being connected to the printer connector. This means that they can only be operated as output lines, but they do have a much larger drive capacity than do unbuffered lines. CA1 can be used directly as described in the general section on 6522s, but note that it is connected to +5 volts via a 4K7 resistor. CA1 normally acts as an 'acknowledge' line when a printer is used. CA2 is buffered so that it has become an *open collector* output only. It usually acts as the printer STROBE line. Note that CA2 can be connected directly to the edge connector using link option 1, see Appendix G.

PORT B — The user port

All of port B lines, i.e. PB0-PB7 and CB1, CB2 are available directly on the user port connector. Figure 23.29 below illustrates the connector. The view is shown looking into the board mounted connector from outside. Note that wires 1 and 20 are the two outermost wires on the ribbon cable. The 'female' part to the connector is a standard 20 way IDC connector. IDC means 'insulation displacement connector'. The plug is normally connected to users' circuits via a length of special 20 way ribbon cable which used to be available from most good computing shops, however walk into a PC superstore nowadays and they wouldn't have a clue what IDC is, let alone sell you a plug! Maplins on the high street, or RS/Farnell are a good source for these items. This cable can be connected to a circuit directly by soldering the wires to the circuit board, or indirectly by another IDC plug and header or a DIL header. A DIL header will plug into any ordinary integrated circuit socket.

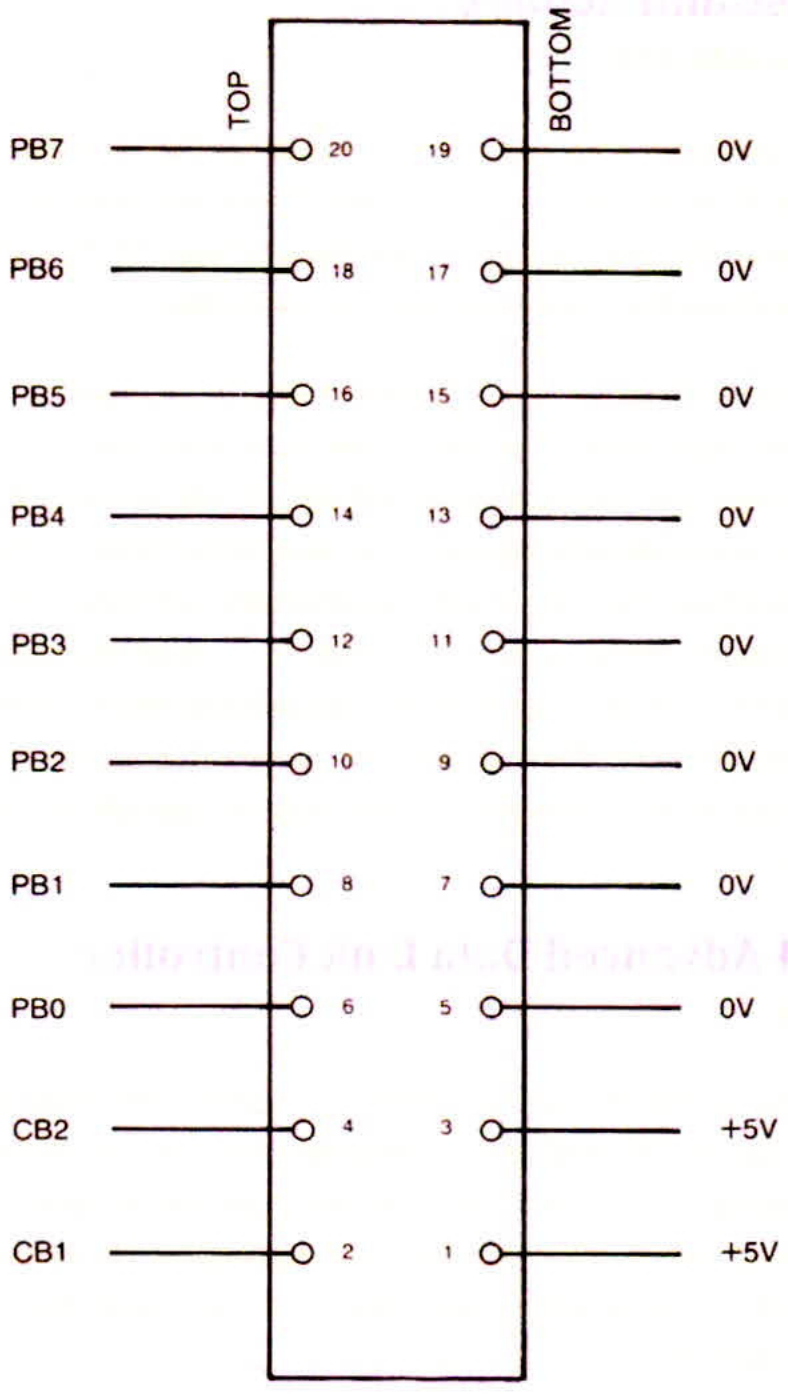


Figure 23.29 – User port connector, looking into socket mounted on the main circuit board.

NOTE: Pins 1 and 20 connect to the wires at the edge of the ribbon cable connected to the IDC header.

Floppy Disc and Econet

Sheila addresses &80-&BF

A floppy disc controller chip and its associated hardware must be fitted to the standard model B before discs can be used. There are several types of upgrade available, including the original 8271 single density type, 1770/1772 double density types, and others based on more obscure disc controllers.

The function of any of these chips is to extract data from a disc or to write data to a disc. Many other tasks have to be performed to ensure that this one basic task is carried out properly. For example, they will detect read errors, refuse to write to a 'protected' disc, automatically position the read/write head on the disc drive plus much more. The chips themselves are exceptionally complex, but luckily there are several very powerful filing system and OSBYTE / OSWORD options available to communicate with them at a higher level than sending bits to control registers and examining results bit by bit. Refer to the Disc System User Guide and Advanced Disc System User Guide that is relevant to your type of upgrade for more information about using discs.

The 68B54 Advanced Data Link Controller

Sheila &A0-&BF

The 68B54 ADLC is the central component in the Econet Interface circuit. In an Econet system up to 255 BBC microcomputers can be connected together. The advantage of doing this is that they may all share (what used to be) expensive peripheral devices such as discs and printers. Nowadays, an Econet is more useful as a fun network for home enthusiasts. Refer to the Econet Manual and OSBYTES &C9, &CE, &CF, &D0 for more information.

The Econet station ID register

Sheila &20 Read only

This will only be valid for users on an Econet system. Reading from this register will return the station ID number. This is set via links S11 to any number between 0 and 255. The Econet data link controller circuit produces NMIs to the CPU. These interrupts are automatically enabled by the hardware every time when the station ID is read.

The Analogue to Digital converter

Sheila addresses &C0-&C2

The analogue to digital converter (ADC) chip provided in the BBC microcomputer is a 10 bit integrating converter. It has four input channels which can be selected under software control. By applying a voltage of between 0 volts and Vref to the channel inputs, a 10 bit binary number will be generated which is directly proportional to the applied voltage. For example applying a voltage Vref/2 would produce a 10 bit value of approximately 511. Vref itself corresponds to about 1023.

Programming the Analogue to Digital converter

The analogue to digital conversion is initiated by writing to the Data Latch/AD start register at Sheila &CO. Bits D1 and D0 together define which one of the four input channels is selected. Bit D3 defines whether an 8 bit resolution or a 10 bit resolution conversion should occur. If set to 0, an 8 bit conversion occurs, if set to 1 a 10 bit conversion occurs. 8 bit conversions typically take 4 ms to complete whereas 10 bit conversions typically take 10 ms to complete. Unless high resolution is required, it is often better to use the fast 8 bit conversion. If enabled, an interrupt will be generated when the conversion is complete. This indicates that valid data can be read from the ADC.

Channel Summary:

Bit 1 control	Bit 0 control	Channel number in MOS and BASIC	Designation
			Master Joystick
0	0	1	Left/Right (low = right)
0	1	2	Up/Down (low = down)
			Secondary Joystick
1	0	3	Left/Right (low = right)
1	1	4	Up/Down (low = down)

Relevant OSBYTES which write to the ADC are &10, &11 and &BD.

Writing to the ADC

There is one register in the analogue to digital converter which can be written to.

Data latch and conversion start - Sheila &C0 Write only

Writing to this register will select the current input channel and select an 8 or 10 bit conversion. The operation of writing to this register automatically initiates a conversion.

Bit 0 and	Define the input channel as shown in
Bit 1	the table above
Bit 2	Flag input, normally set to 0
Bit 3	8 bit mode = 0
	10 bit mode = 1
Bits 4-7	not used

Reading from the ADC

There are three registers in the ADC which can be read directly, the status register and two data registers.

Status Register - Sheila &C0 Read only

Bit 0 and	These define the currently selected input channel
Bit 1	as in the AD start register.
Bit 2	not used
Bit 3	8 bit mode = 0
	10 bit mode = 1
Bit 4	2nd most significant bit (MSB) of conversion.
Bit 5	MSB of conversion.
Bit 6	0 = busy, 1 = not busy
Bit 7	0 = conversion completed
	1 = conversion not completed

High data byte - Sheila &C1 Read only

This byte contains the 8 most significant bits of the analogue to digital conversion.

Low data byte - Sheila &C2 Read only

Bits 7 to bit 4 define the four low order bits of a 12 bit conversion. In 8 bit only mode, all four bits are inaccurate. In 10 bit mode, bits 7 and 6 are accurate. Bits 5

and 4 are likely to be inaccurate but this will depend upon the particular qualities of individual 7002 chips. Bits 3—0 are always set to low.

OSBYTES &80, &BC, &BD and &BE are relevant when reading from the ADC.

Hardware connections for a joystick

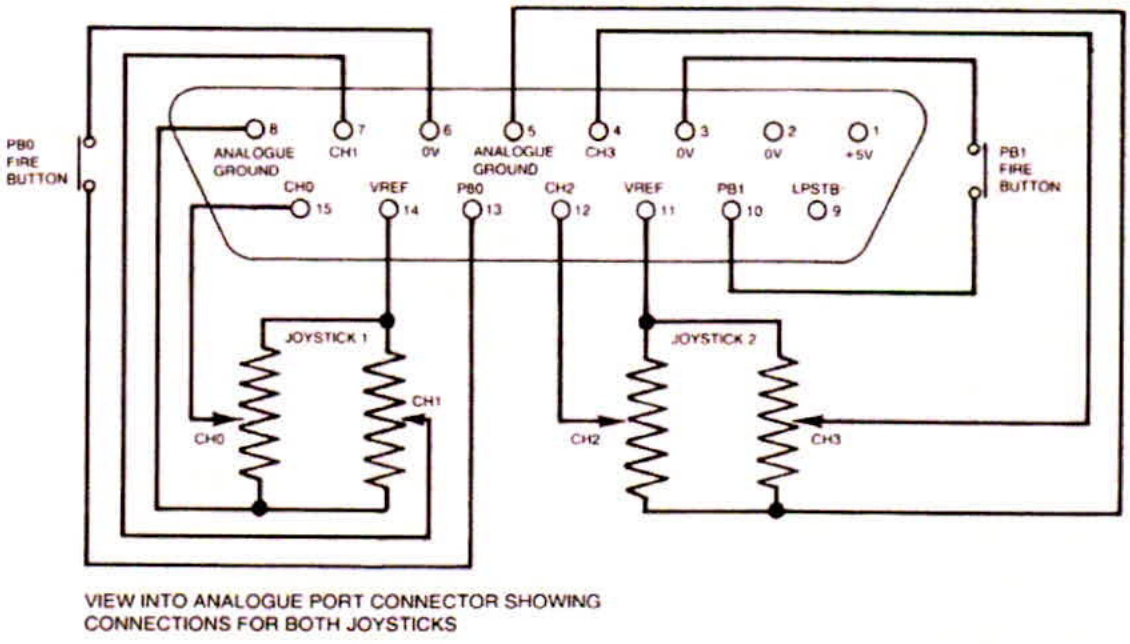


Figure 23.30 – Connections for Joysticks

A 15 way 'D' type connector is provided on the rear of all model Bs. This is the analogue port connector. The pin designations and layout are illustrated in figure 23.30. Connections required for the various joysticks are also illustrated. Note the two 'fire' buttons are there as well. Their states can be determined using OSBYTE &80.

Apart from giving the information needed to construct joysticks, this diagram should be helpful to anyone wishing to be a bit more adventurous with their hardware. One use for the full 10 bits available would be communication with a graphics tablet. By using high-quality variable resistors and a series of pulleys or lever arms, the position of a pointer can be determined. This would allow high resolution graphic drawing to be entered into the BBC microcomputer for display on the high resolution screen. Some other possibilities include measuring temperature, light level, pH (hydrogen ion concentration), current, voltage, resistance or pressure.

The One Megahertz bus

There are basically two routes which a user can take towards adding his own hardware. One of these is the 6522 USER port. The problem with the USER port is that there are only 8 I/O lines and a couple of control lines. For more complex peripherals, direct access to the 6502 address and data buses are required. This interface is provided by the one megahertz bus.

Physically, the one megahertz bus interface is a 34 pin connector mounted at the front edge of the main BBC microcomputer circuit board. It is accessed from underneath the keyboard. A buffered databus and the lower 8 bits of the address bus are connected to this socket together with a series of useful control signals. Whilst the designer could use the one megahertz bus in innumerable different configurations, Acorn has defined how the bus should be used to maintain compatibility with other devices.

The standard uses of the one megahertz bus allow up to 64K bytes of paged memory to be used as well as 255 direct memory mapped devices (plus the paging register). 'FRED' is normally assigned as the memory mapped I/O page and 'JIM' is normally assigned as the 64K memory expansion page. Communication between FRED, JIM and programs should be implemented using OSBYTEs &92, &93, &94 and &95.

'FRED' and Memory Mapped Hardware

Page &FC in the BBC microcomputer is reserved for peripherals with small memory requirements. The initial allocations of space in FRED are:

&FC00 - &FC0F	Test Hardware
&FC10 - &FC13	Teletext
&FC14 - &FC1F	Prestel
&FC20 - &FC27	IEEE 488 Interface
&FC28 - &FC2F	Acorn Expansion, currently unused
&FC30 - &FC3F	Cambridge Ring Interface
(&FC20-&FC3C	BeebSID (M.Barr))
&FC40 - &FC4F	Compact Flash Interface (&FC48 for 16 bit interfaces)
&FC80 - &FC8F	Test Hardware
&FC90 - &FCBF	Acorn Expansion, currently unused
&FCC0 - &FCF7	User Applications
&FCF8 - &FCFD	RetroClinic DataCentre
&FCFE - &FCFF	16 bit paging register for JIM

When designing circuits to add on to the one megahertz bus, the 'Not page &FC' (NPGFC) signal together with the lower 8 address lines should be decoded to select the add-on circuit.

NOTE: A 'clean up' circuit will be required on the NPGFC signal in most applications.

For very keen constructors who require more than the 55 page &FC locations reserved for User Applications, either page &FD can be used for memory mapped peripherals or other FRED locations can be used. Using reserved FRED locations in this way will mean that the hardware add-ons specified for those locations cannot be added in future if user hardware is already using the slot.

'JIM' and 64K (8 bit) or 16Mb (16 bit) Paged Memory

Page &FD in the BBC microcomputer address space is used in conjunction with the paging register in FRED to provide an extra bank of memory. Using just &FCFF as an 8 bit paging register will give access to 64K or additional RAM. Using it as a 16 bit register, with the MSB at &FCFE, 16 Megabytes of RAM can be made available.

This memory is accessed one page at a time. The particular page being accessed is selected by the value in FRED's paging register(s), and is referred to as the

'Extended page number'. Note that a 'Not page &FD' (NPGFD) signal is available on the one megahertz bus connector. Accessing memory through the 1MHz bus will generally be about twenty times slower than accessing main memory if using OSBYTE calls, however direct access is only about half the speed.

Extended page allocation

'Extended pages' &00 - &7F in JIM are reserved for use by Acorn. The other pages &80 - &FF are reserved for user applications.

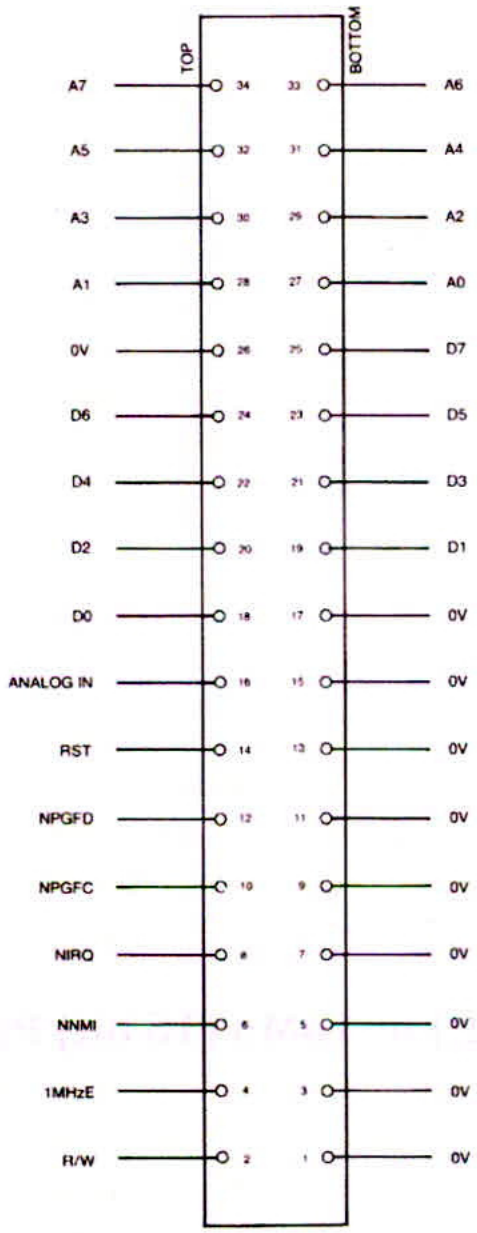


Figure 23.31 – The 1MHz bus connector, looking into socket mounted on main board

NOTE: Pins 1 and 34 connect to wires at the edge of the ribbon cable.

Bus signal definitions

The one megahertz bus connector is illustrated in Figure 23.31. The specification for the signals on the one megahertz bus are:

0 volts — This is connected to the main system 0 volts line. The reason for putting 0V lines between the active signal lines is to reduce the interference between different signals.

R/W (pin 2) — This is the read-not-write signal from the 6502 CPU, buffered by two 74LS04 inverters.

1MHzE (pin 4) — This is the 1MHz system timing clock. It is a 50% duty-cycle square wave. The 6502 CPU is operating at 2MHz, so the main processor clock is stretched whenever 1MHz bus peripherals are being accessed. The trailing edges of the 1MHzE and 2MHz processor clock are then coincidental. The processor clock is only ever truly 2MHz when accessing main memory.

NNMI (pin 6) — Not Non-Maskable Interrupt. This is connected directly to the 6502 NMI input. It is pulled up to +5 volts with a 3K3 resistor. Use of Non-Maskable Interrupts on the BBC microcomputer is only advisable after Chapter 19 on interrupts has been read and thoroughly understood. Both Disc and Econet systems rely heavily upon NMIs for their operation so take care. Note that NMIs are triggered on negative going edges of NMI signals.

NIRQ (pin 8) — Not Interrupt Request. This is connected directly to the 6502 IRQ input. Any devices connected to this input should have *open collector* outputs. The line is pulled up to +5 volts with a 3K3 resistor. Interrupts from the 1MHz bus must not occur until the software on the main system is able to cope with them. All interrupts must therefore be disabled after a reset. Note that the main system software may operate very slowly if considerable use is made of interrupts. Certain functions such as the real time clock which is incremented every 10 mS will be affected if interrupts are masked for more than this period. Refer to Chapter 19 on interrupts for more information.

NPGFC (pin 10) Not page &FC. This signal is derived from the 6502 address bus. It goes low whenever page &FC is written to or read from. FRED is the name given to this page in memory and it is described in more detail above.

NPGFD (pin 12) Not page &FD. This signal is derived from the 6502 address bus. It goes low whenever page &FD is accessed. JIM is the name given to this page in memory.

NRST (pin 14) Not RESET. This is an active low output from the system reset line. It may be used to initialise peripherals whenever a power up or pressing <BREAK> causes a reset.

Analogue Input This is an input to the audio amplifier on (pin 16) the main computer. The amplified signal is produced over the speaker on the keyboard. Its input impedance is 9K Ohms and a 3 volt RMS signal will produce maximum volume on the speaker. Note however that signals as large as this will cause distortion if the sound or speech is used at the same time.

D0 - D7 (pins 18 - 24) This is a bi-directional 8 bit data bus which is connected via a 74LS245 buffer (IC72) to the CPU. The direction of data transfer is determined by the R/W line signal. The buffer is enabled whenever FRED or JIM are accessed.

A0-A7 (pins 27 - 34) These connected directly to the lower 8 CPU address lines via a 74LS244 buffer (1C71) which is always enabled.

'Cleaning up' FRED and JIM's page selects

All 1MHz peripherals are clocked by a 1MHz 50% duty cycle square wave, designated as 1MHzE in Figure 23.32. This clock rate was chosen to allow chips such as 6522 VIAs to use their internal timing elements correctly. The system 6502 CPU is normally clocked at twice the speed of the peripherals and so it operates at 2MHz. However, if the CPU wishes to access any device on the 1MHz bus, the processor has to be slowed down. The effect of this slow down circuit is illustrated in Figure 23.32. After generating a valid 1MHz address, the slow down circuit stretches the clock high period (from 'T' to 'U'). Unfortunately, two major problems arise from this mode of operation.

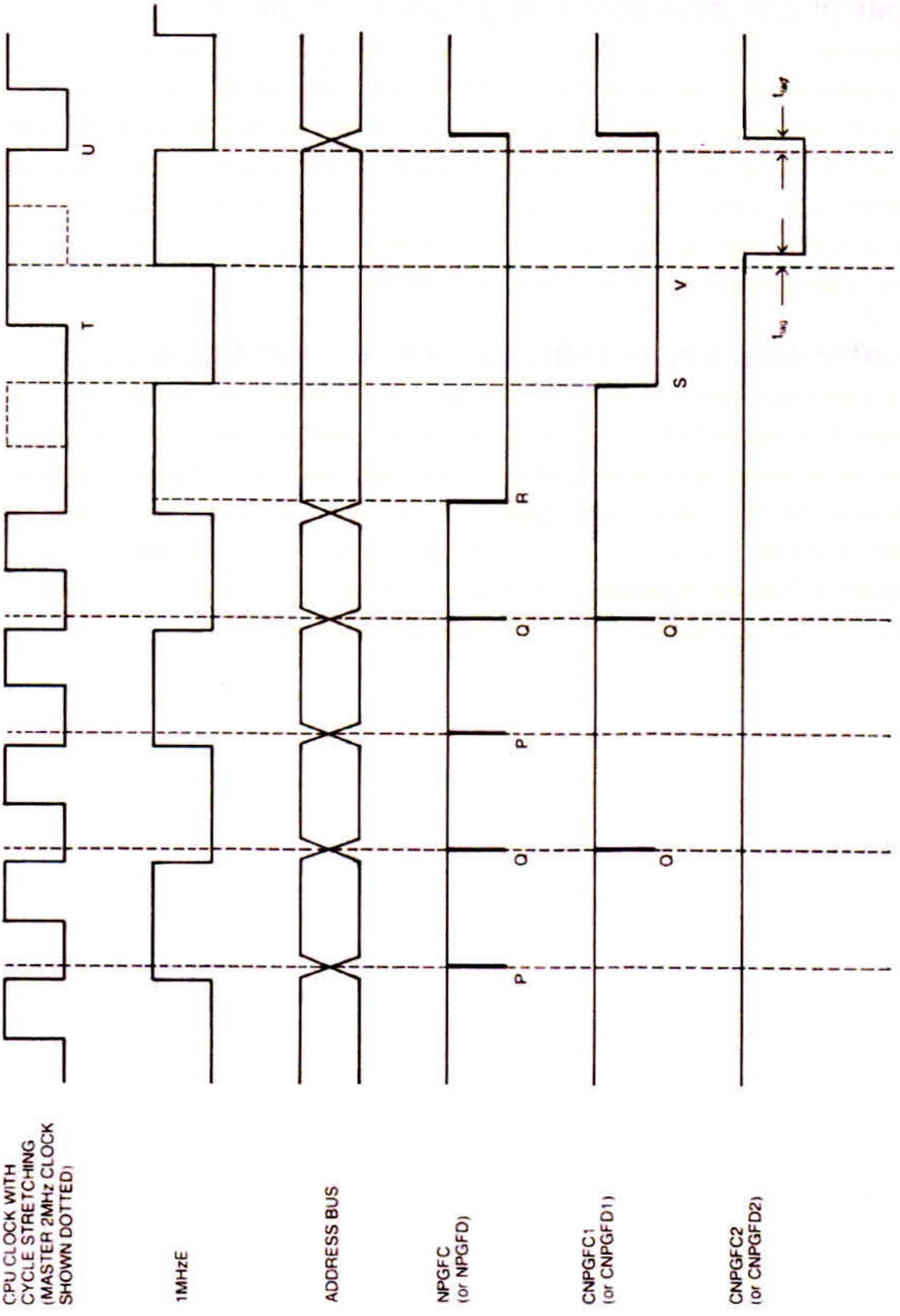


Figure 23.32 – 1MHz bus timing showing page select signals

Spurious address decoding 'glitches' - PROBLEM 1

Addresses on the system address bus will only usually change when the 2MHz processor clock is low. However, the 1MHz clock is alternately low, then high when the CPU addresses change. This gives rise to the address decoding glitches labelled 'P' and 'Q' in figure 23.32. The 'Q' glitches are not normally important because the 1MHzE clock is then low. The 'P' glitches can cause problems because the 1MHzE signal is then high. Spurious pulses may therefore occur on the various chip select pins, leading to possible malfunction of some devices.

Double accessing of 1MHz bus devices - PROBLEM 2

If a 1MHz bus device is accessed during a period when the 1MHzE clock is high (point 'R' in figure 23.32), that device will be accessed immediately. The device will then be accessed again when 1MHzE is next high (point 'V' in figure 23.32). This is because the CPU clock is held high until the next coincident falling edge of the 2MHz and 1MHz clocks (point 'U'). Double accessing a peripheral does not normally present a problem. However, if reading from or writing to a device has some other function, such as clearing an interrupt flag, a problem may occur.

'Clean up' circuit 1

This standard 'clean up' circuit for the page select signals is shown in Figure 23.33. Three NOR gates are used to create a standard R-S flip-flop with a gated input. The 'clean page select' output (CNPGFC1) can only be set low if 1MHzE is low. The net effect of the circuit is illustrated in figure 23.32. Both of the problems outlined above are overcome, since the 'P' glitches are removed and the page select only goes low at '5', after the 1MHzE clock has gone low. The 'Q' glitches due to spurious addresses whilst 1MHzE is low are still there. In most applications, this will not affect circuit operation, but occasionally a totally glitch free page select will be required. Circuit 2 will provide this type of page select.

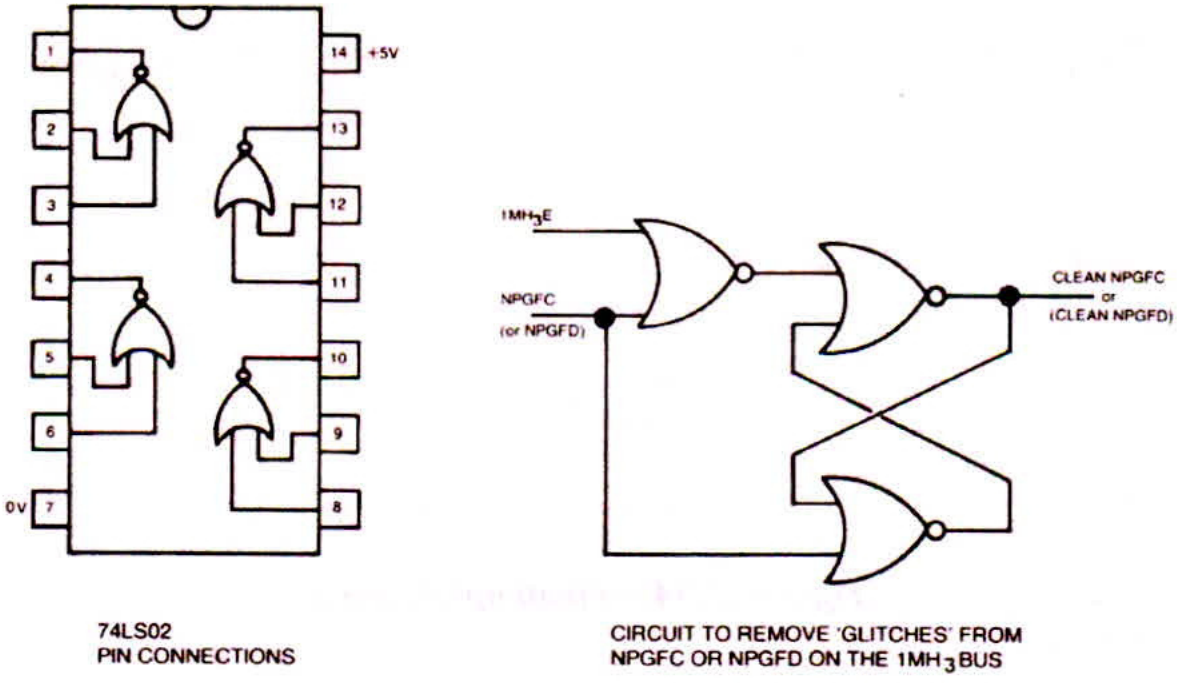


Figure 23.33 – 'Clean up' circuit 1

'Clean up' circuit 2

In situations in which a 100% 'clean' page select signal is required, circuit 2 which is illustrated in Figure 23.34 should be used. Before CNPGFC can go low, a valid page address with 1MHzE low must occur. The page low is then latched into a D-type flip-flop on the rising edge of the 1MHzE clock. As shown in Figure 23.32, CNPGFC2 will go low a time lag (40nS) after 1MHzE goes high and it will remain valid until 40nS after 1MHzE has gone low again. Take care if any of the lines on the 1MHz bus are buffered, because delays introduced by buffers could make data invalid when it is latched. Refer to the precise timing diagram (Figure 23.35) for more details.

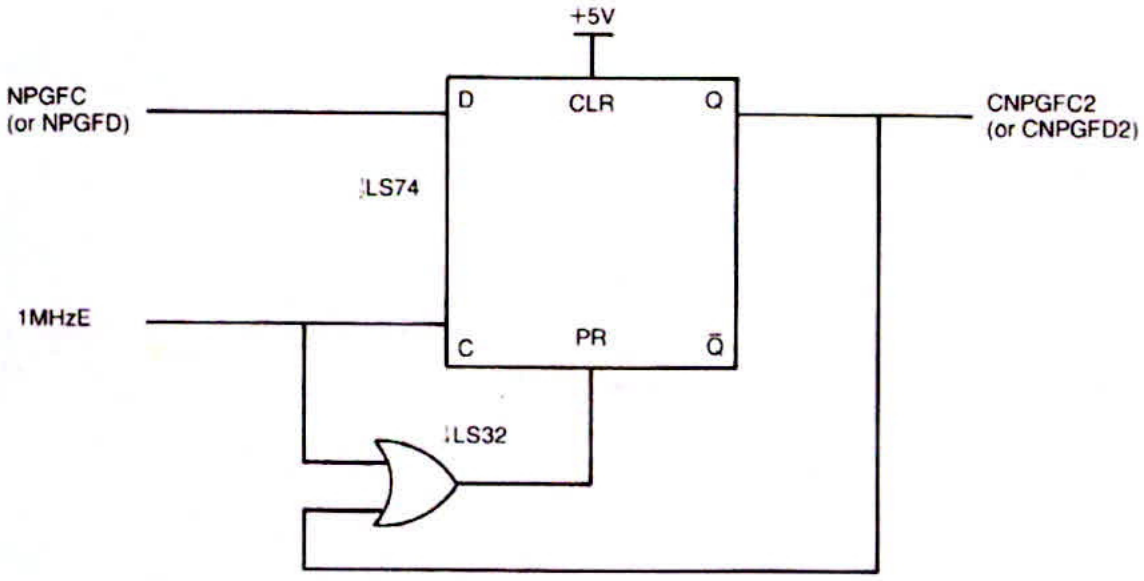


Figure 23.34 – 'Clean up' circuit 2

Hardware requirements for 1MHz bus peripherals

All additional hardware designed to operate from the 1MHz bus must conform to the following standards:

Power supply

No power should be drawn from the BBC microcomputer. All peripherals should have their own integral power supply, or use a separate power supply unit.

Logic line loading

No more than one low power Schottky TTL load should be presented to any of the logic lines by a peripheral. In most instances, this means that all logic lines will have to be buffered for each peripheral.

Connection to the BBC microcomputer

Connection to the BBC microcomputer should be via a 600mm length of 34-way ribbon cable terminated with a 34-way IDC socket. The 1MHz bus connections should 'feed through' the unit, i.e.. a 34-way output header plug connector should be provided so that more devices can be connected as required.

Bus termination

All bus lines except NRST, NNMI and NIRQ should be provided with the facility for adding optional termination. The recommended way of terminating lines is to connect each one to +5V with a 2K2 resistor and to 0V with a 2K2 resistor.

Timing requirements

The 1MHz bus timing requirements are illustrated in the timing diagram, Figure 23.35. It should be noted that these timings are based on the assumption of only one peripheral being attached to the bus. Heavier loading may extend the *rise* and *fall* times of 1MHzE with possible adverse effects on timings.

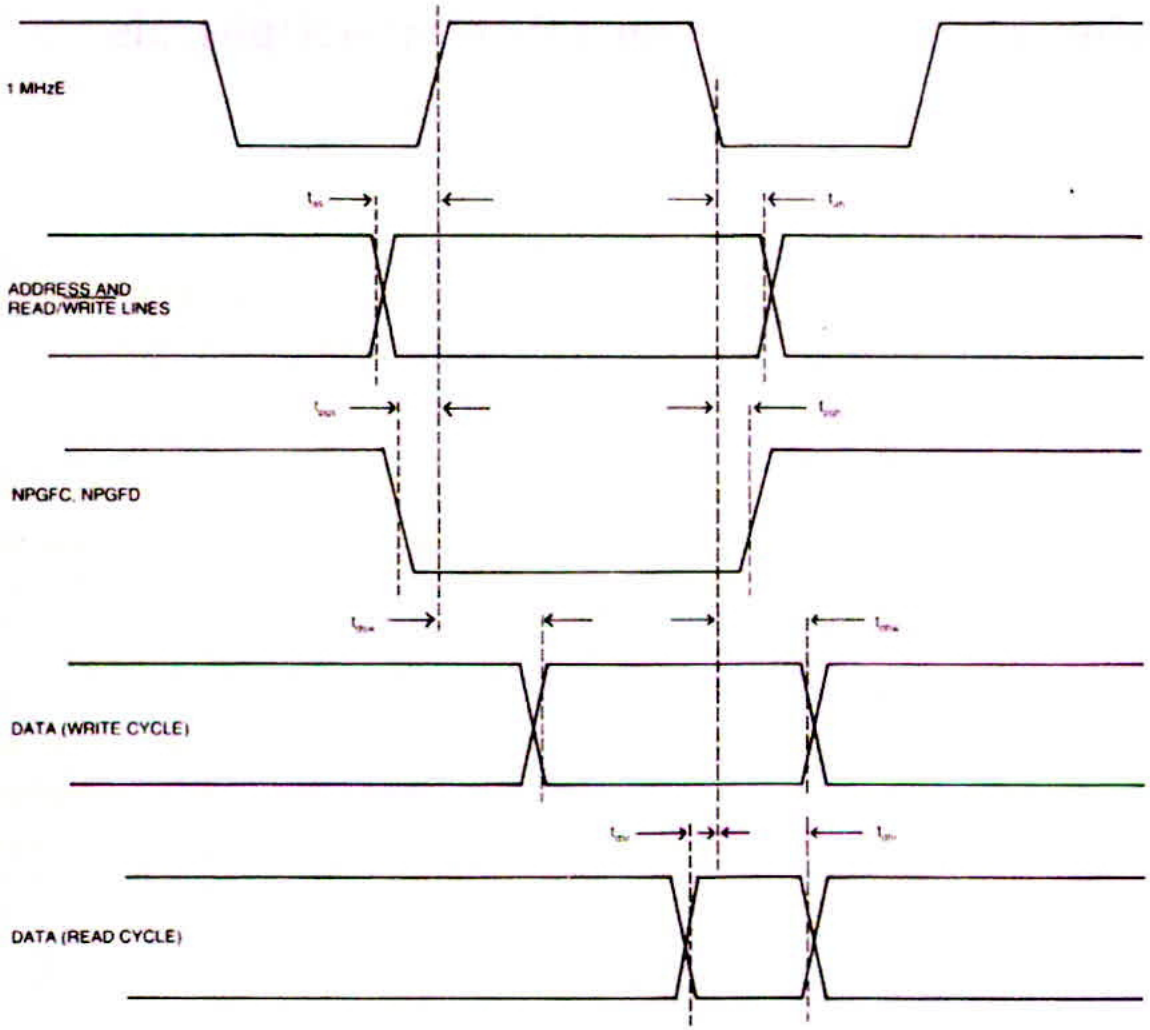


Figure 23.35 – 1MHz bus timing

The timing requirements are:

Description	Symbol	Min.	Max.
Address (and Read/Write) Set-up time	t as	300nS	1000nS
Address (and Read/Write) Hold time	t ah	30nS	-
NPGFC & NPGFD Set-up time	t pgs	250nS	1000nS
NPGFC & NPGFD Hold time	t pgh	30nS	-
Write data set-up time	t dsw	-	150nS
Write data hold time	t dhw	50nS	-
Read data set-up time	t dsr	200nS	-
Read data hold time	t dhr	30nS	-

The Tube

Sheila addresses &E0-&FF

The BBC microcomputer is provided with three **BASIC** methods of expansion. The user port and the one megahertz bus are covered in previous sections of this Chapter. This section covers the third expansion route - the Tube. The Tube itself is just a fast parallel communication link between two computers. First of all, the basic fundamentals required of any Tube system are explained followed by some specific examples of its use with different processors.

The Tube connector on the BBC microcomputer simply consists of the system data bus, the lower half of the system address bus and some miscellaneous control lines. These connect via a ribbon cable to a *second processor* card. The second processor is connected to the Tube interface by a Tube ULA (uncommitted logic array) chip which has been designed specifically for this application by Acorn.

The Tube ULA provides a completely asynchronous parallel interface between the two processor systems. The original BBC microcomputer is the 'HOST' system and the new second processor forms the heart of the 'PARASITE' system. To each system, the Tube resembles a conventional peripheral device which occupies 8 bytes of memory or I/O space. Four bytewide read only and four byte-wide write only latches are provided within this address space, together with their associated control registers.

The byte-wide communication paths fall into two distinct categories. The first set are simply latches, so data written in on one side is read out directly on the other side. The second set are FIFO (first in first out) buffers which store two or more bytes at a time. These bytes can then be read out on the other side of the Tube in the same order that they were entered into the buffer.

Data and messages are passed back and forth through the various registers according to carefully designed software protocols. Proper allocation of the registers to specific tasks allows both systems to operate at maximum efficiency. For example, complex VDU plot and colour fill commands can be sent via the largest FIFO from the *parasite* to the *host* processor. The parasite processor will not then have to wait for the host to finish processing the laborious VDU commands before continuing with its language processing.

Some second processors and their uses

There are several second processors can be used on the Tube. Amongst these are a number of 6502 types, a Z80, 16032, 68000 "Casper", and an ARM evaluation system. Other second processors that can be fitted using external adaptors are the 80186 Master 512, and the ARM-7, from Sprow.

The 6502 Second processor

The standard Acorn 6502B on the Tube is a faster 3MHz version of the 2Mhz 6502A used in the main BBC microcomputer. It is provided with a full 64K of RAM. When the machine is powered up, the default language is copied across from the main BBC to the Tube processor. From then on the main BBC microcomputer 6502A is turned into a servant. Its purpose is that of handling all input from the RS423, keyboard, joysticks etc. and output to the screen, sound generator etc. It sends its input across the Tube to the fast 6502B and executes laborious tasks which the fast 6502B sends back. All of the processing for languages or applications packages are performed by the Tube processor. The advantage is that the work load is shared out. Because the fast 6502B doesn't have to worry about interrupts from any of the devices connected to a BBC microcomputer, it can process at a very fast speed. The old 6502A does all output like plotting graphics on the screen, but it doesn't ever need to do any language processing. Programs will generally run at almost twice their original speed.

If the above process sounds rather complex, then don't worry. BBC **BASIC** will appear to operate almost exactly as normal over the Tube. The major difference will be the extra memory available. Since all of the screen memory resides in the *host* processor, the *parasite* has all of its memory available for program and data storage. Naturally, 16K of the *parasite's* RAM will be required for **BASIC**, but most of the remaining 48K is available to the programmer (compared with a maximum of 27K on an ordinary BBC microcomputer). The 16K operating system ROM stays in the *host* processor's memory map and is not transferred across the Tube.

Other 6502 second processors have come to market which use faster versions of the 6502, such as the ReCoPro by John Kortink, and the NuCoPro, which is currently in development.

The Z80 Second processor

Like the 6502, the Z80 is a microprocessor, but with a different instruction set. It can also operate over the Tube. The set up is still very similar to that for a 6502 since the Z80 does all language processing and the BBC microcomputer 6502 does all of the I/O processing. Machine code programs written to run on the BBC micro will not operate with a Z80 Tube. However, the vast amount of Z80 software which is available will operate on a Z80 Tube machine. The operating system called CP/M is supplied as standard with all Z80 second processors. There is a large quantity of CP/M software (business packages, most languages, games + almost everything else) available on other CPM machines. Some of this software will operate on the BBC microcomputer with a Z80 second processor, provided that the disc format is correct.

Torch also do a series of Z80 co-processors for the BBC Microcomputer system.

The 32016 (Cambridge) Second processor

As with the 6502 and Z80 Tubes, the old BBC micro 6502 still does all input/output and the 32016 runs languages. Unlike the 6502 and Z80 which are both relatively old 8 bit processors, the 32016 is one of a new generation of 16 bit processors. Its internal structure operates on 32 bits, and it has a very nicely organised instruction set which is very powerful. With one of these sitting on the end of the Tube, and a Hard Disc Drive connected to the *host* processor, the computing power available will be equal that available on many mainframe computers. One standard operating system provided on 32016 Tube machines is PanOS. The 32016 was later renamed to the Cambridge Co-Processor, to match the name used by the Acorn Cambridge Workstation.

68000 "Casper"

This was a little known co-processor which fitted inside the case of the BBC. It is programmable in assembler, and has 128K of ROM and 128K of RAM.

Appendix

Appendix A - *FX/OSBYTE call index

	Brief description	Dec.	Hex.
***CODE**	perform *CODE	136	88
***MOTOR**	perform *MOTOR	137	89
***OPT**	perform *OPT	139	8B
***ROM**	perform *ROM	141	8D
***TAPE**	perform *TAPE	140	8C
***TV**	perform *TV	144	90
ADC	select channels sampled	16	10
	force ADC conversion	17	11
	read ADC channel	128	80
	read current ADC channel	188	BC
	read/write max channel number	189	BD
	read ADC conversion type	190	BE
BELL/ctrl G	read/write BELL channel number	211	D3
	read/write BELL envelope number	212	D4
	read/write BELL frequency	213	D5
	read/write BELL duration	214	D6
BREAK/reset	read/write critical region	200	C8
	read/write message + !BOOT opts	215	D7
	read/write BREAK intercept code	247	F7
	+	248	F8
	+	249	F9
	read/write last BREAK type	253	FD
	read/write start up options	255	FF
buffer	flush selected class	15	0F
	flush particular buffer	21	15
	get buffer status	128	80
	insert value into buffer	138	8A
	get character from buffer	145	91
	examine buffer status	152	98
	insert char. into I/P buffer	153	99

	Brief description	Dec.	Hex.
bus	read from FRED	146	92
	write to FRED	147	93
	read from JIM	148	94
	write to JIM	149	95
	read from SHEILA	150	96
	write to SHEILA	151	97
cursor	enable cursor editing	4	04
	read text cursor position	134	86
	read character at cursor	135	87
	read/write cursor editing state	237	ED
Econet	read/write net keyboard disable	201	C9
	read/write OS call intercept	206	CE
	read/write OSRDCH intercept	207	CF
	read/write OSWRCH intercept	208	D0
ESCAPE	clear ESCAPE condition	124	7C
	set ESCAPE condition	123	7D
	acknowledge ESCAPE	126	7E
	read/write BREAK, ESCAPE effects	200	C8
	read/write ESCAPE key value	220	DC
	read/write ESCAPE key status	229	E5
	read/write ESCAPE effects flag	230	E4
Events	disable event	13	0D
	enable event	14	0E
explode	character definition RAM	20	14
	read/write explosion state	182	B6
Files	close EXEC or SPOOL files	119	77
	check for EOF of open file	127	7F
	fast Tube BPUT	157	9D
	read/write CFS timeout counter	176	B0
	*TAPE/*ROM switch	183	B7
	read/write EXEC file handle	198	C6
	read/write SPOOL file handle	199	C7
Flashing	flashing col. mark duration	9	09
	flashing col. space duration	10	0A
	read/write flash counter	193	C1
	read/write mark period count	194	C2
	read/write space period count	195	C3

	Brief Description	Dec.	Hex.
Input	select I/P stream	2	02
	read/write input source	177	B1
keyboard	auto—repeat delay	11	0B
	auto—repeat rate	12	0C
	reflect keyboard status in LEDs	118	76
	write current keys pressed	120	78
	perform keyboard scan	121	79
	perform keyboard scan from 16	122	7A
	read key with time limit	129	81
	read key translate address	172	AC
	+	173	AD
	read/write keyboard semaphore	178	B2
	read/write auto—repeat delay	196	C4
	read/write auto-repeat period	197	C5
	read/write net keyboard disable	201	C9
	read/write keyboard status	202	CA
	read/write TAB key value	219	DB
	read/write ESCAPE key value	220	DC
Memory	character definition RAM	20	14
	read high order address	130	82
	read top of OS RAM (OSHWM)	131	83
	read bottom of display (HIMEM)	132	84
	read hypothetical HIMEM	133	85
	read/write primary OSHWM	179	B3
	read/write current OSHWM	180	B4
	read/write ch. explosion state	182	B6
	read/write ESCAPE, BREAK effects	200	C5
	read/write location &280	240	F0
	read/write user OSBYTE location	241	F1
	read/write location &28A	250	FA
	read/write location &28B	251	FB
	read/write available RAM	254	FE
O.S.	version number	0	00
	read address of OS variables	166	A6
	+	167	A7
Output	select O/P stream	3	03
	read/write O/P status	236	EC

	Brief description	Dec.	Hex.
paged ROM	enter language ROM	142	8E
	issue ROM service request	143	8F
	read address of ROM pointers	168	A8
	+	169	A9
	read address of ROM information	170	AA
	+	171	AB
	read ROM active at last BRK	186	BA
	BASIC ROM number	187	BB
	read/write current language no.	252	FC
printer	printer destination	5	05
	set printer char. ignored	6	06
	printer driver going dormant	123	7B
	read/write printer destination	245	F5
	read/write character ignored	246	F6
RS423	RX baud rate	7	07
	TX baud rate	8	08
	read/write RS423 mode	181	B5
	read /write RS423 use flag	191	BF
	read RS423 control flag	192	C0
	read/write RS423 handshake	203	CB
	read/write I/P suppression flag	204	CC
	read/write RS423/cassette select	205	CD
Serial	read/write 6850 control reg.	156	9C
	read/write IRQ mask, 6850	232	E5
	read RAM copy of serial ULA reg	242	F2
soft keys	disable cursor editing	4	04
	reset soft keys	18	12
	read/write key string length	216	D8
	read/write &C0 to &CF status	221	DD
	read/write &D0 to &DF status	222	DE
	read/write &E0 to &EF status	223	DF
	read/write &F0 to &FF status	224	E0
	read/write function key status	225	E1
	read/write SHIFT fn key status	226	E2
	read/write CTRL fn key status	227	E3
	read/write CTRL+SHIFT fn status	228	E4
	read/write consistency flag	244	F4

	Brief description	Dec.	Hex.
sound	read/write sound suppression	210	D2
speech	read from speech processor	158	9E
	write to speech processor	159	9F
	read/write speech suppression	209	D1
	read speech presence	235	EB
Timer	read/write timer switch state	243	F3
Tube	fast Tube BPUT	157	B9
	read Tube presence	234	EA
user	user OSBYTE call	1	01
	read/write user OSBYTE location	241	F1
vertical sync	wait for vertical sync	19	13
VDU	read VDU status	117	75
	read VDU variable value	160	A0
	read address of VDU variables	174	AE
	+	175	AF
	read/write VDU queue length	218	D8
VIA/6522	read/write IRQ mask, user VIA	231	E7
	read/write IRQ mask, system VIA	233	E9
video	flashing col. mark duration	9	09
	flashing col. space duration	10	0A
	wait for vertical sync	19	13
	select shadow screen mode (OS2.00+ only)	114	72
	write to ULA palette register	154	9A
	write to ULA control register	155	9B
	read RAM copy of ULA ctrl reg.	184	B8
	read RAM copy of ULA pal. reg.	185	B9
	read/write flash counter	193	C1
	read/write mark period count	194	C2
	read/write space period count	195	C3
	read/write lines from page halt	217	D9

Appendix B - Operating System calls summary

Routine Name	Address	Vector Name	Vector Address	Summary of function
		USERV	200	The user vector
		BRKV	202	The BRK vector
		IRQ1V	204	Primary interrupt vector
		IRQ2V	206	Unrecognised IRQ vector
OSCLI	FFF7	CLIV	208	Command line interpreter
OSBYTE	FFF4	BYTEV	20A	*FX/OSBYTE call
OSWORD	FFF1	WORDV	20C	OSWORD call
OSWRCH	FFEE	WRCHV	20E	Write character
OSNEWL	FFE7	—	—	Write LF, CR to screen
OSASCI	FFE3	—	—	Write character &0D=LF
OSRDCH	FFE0	RDCHV	210	Read character
OSFILE	FFDD	FILEV	212	Load/save file
OSARGS	FFDA	ARGSV	214	Load/save file data
OSBGET	FFD7	BGETV	216	Get byte from file
OSBPUT	FFD4	BPUTV	218	Put byte in file
OSGBPB	FFD1	GBPBV	21A	Multiple BPUT BGET
OSFIND	FFCE	FINDV	21C	Open or close file
		FSCV	21E	File system control entry
		EVNTV	220	Event vector
		UPTV	222	User print routine
		NETV	224	Econet vector
		VDUV	226	Unrecognised VDU commands
		KEYV	228	Keyboard vector
		INSV	22A	Insert into buffer vector
		REMV	22C	Remove from buffer vector
		CNPV	22E	Count/purge buffer vector
		IND1V	230	Spare vector
		IND2V	232	Spare vector
		IND3V	234	Spare vector
NVRDCH	FFCB	—	—	Non—vectored read char.
NVWRCH	FFC8	—	—	Non—vectored write char.
GSREAD	FFC5	—	—	Read char. from string
GSINIT	FFC2	—	—	String input initialise
OSEVEN	FFBF	—	—	Generate an event
OSRDRM	FFB9	—	—	Read byte in paged ROM

Appendix C - Key Values Summary

Key/ character	ASCII		INKEY		Internal Key No.	
	Dec	Hex	Dec	Hex	Dec	Hex
SPACE	32	20	- 99	9D	98	62
!	33	21				
"	34	22				
#	35	23				
$	36	24				
%	37	25				
&	38	26				
'	39	27				
(	40	28				
)	41	29				
*	42	2A				
+	43	2B				
,	44	2C	-103	99	102	66
-	45	2D	-24	E8	23	17
.	46	2E	-104	98	103	67
/	47	2E	-121	87	104	68
0	48	30	-40	D8	39	27
1	49	31	-49	CF	48	30
2	50	32	-50	CE	49	31
3	51	33	-18	EE	17	11
4	52	34	-19	ED	18	12
5	53	35	-20	EC	19	13
6	54	36	-53	CB	52	34
7	55	37	-37	DB	36	24
8	56	38	-22	EA	21	15
9	57	39	-39	D9	37	25
:	58	3A	-73	B7	72	48
;	59	3B	-88	A8	87	57
<	60	3C				
=	61	3D				
>	62	3E				
?	63	3F				
@	64	40	-72	B8	71	47
A	65	41	-66	BE	65	41
B	66	42	-101	9B	100	64
C	67	43	-83	AD	82	52
D	68	44	-51	CD	50	32
E	69	45	-35	DD	34	22
F	70	46	-68	BC	67	43
G	71	47	- 84	AC	83	53
H	72	48	-85	AB	84	54
I	73	49	-38	DA	38	26
J	74	4A	-70	BA	69	45

Key/ character	ASCII		INKEY		Internal Key No.	
	Dec	Hex	Dec	Hex	Dec	Hex
K	75	4B	-71	B9	70	46
L	76	4C	-87	A9	86	56
M	77	4D	-102	9A	101	65
N	78	4E	-86	AA	85	55
O	79	4E	-55	C9	54	36
P	80	50	-56	C8	55	37
Q	81	51	-17	EF	16	10
R	82	52	-52	CC	51	33
S	83	53	-82	AF	81	51
T	84	54	-36	DC	35	23
U	85	55	-54	CA	53	35
V	86	56	-100	9C	99	63
W	87	57	-34	DE	33	21
X	88	58	-67	BD	66	42
Y	89	59	-69	BB	68	44
Z	90	5A	-98	9E	97	61
[	91	5B	-57	C7	56	38
\	92	5C	-121	87	120	78
]	93	5D	-89	A7	88	58
^	94	5E	-25	E7	24	18
_	95	5F	-41	D7	40	28
£	96	60				
a	97	61				
b	98	62				
c	99	63				
d	100	64				
e	101	6S				
f	102	66				
g	103	67				
h	104	68				
i	105	69				
j	106	6A				
k	107	6B				
l	108	6C				
m	109	6D				
n	110	6E				
o	111	6F				
p	112	70				
q	113	71				
r	114	72				
s	115	73				
t	116	74				
u	117	75				
v	118	76				
w	119	77				

Key/ character	ASCII		INKEY		Internal Key No.	
	Dec	Hex	Dec	Hex	Dec	Hex
x	120	78				
y	121	79				
z	122	7A				
{	123	7B				
|	124	7C				
}	125	7D				
~	126	7E				
ESCAPE	27	1B	-113	8F	112	70
TAB	9	09	-97	9F	86	60
CAPS LOCK			-65	BF	64	40
CTRL			-2	FE	1	01
SHIFT LOCK			-81	AF	80	50
SHIFT			-1	FF	0	00
DELETE	127	7F	-90	A6	89	59
COPY			-106	96	105	69
RETURN	13	0D	-74	B6	73	49
UP CURSOR			-58	C6	57	39
DOWN CURSOR			-42	D6	41	29
LEFT CURSOR			-26	E6	25	19
RIGHT CURSOR			-122	86	121	79
f0			-33	DF	32	20
f1			-114	8E	113	71
f2			-115	8D	114	72
f3			-116	8F	115	73
f4			-21	EB	20	14
f5			-117	8B	116	74
f6			-118	8C	117	75
f7			-23	E9	22	16
f8			-19	8D	118	76
f9			-120	8E	119	77
Startup option switch (on front of keyboard)						
bit 0					9	09
bit 1					8	08
bit 2					7	07
bit 3					6	06
bit 4					5	05
bit 5					4	04
bit 6					3	03
bit 7					2	02

Appendix D - VDU Code Summary

Dec	Hex	CTRL	+bytes	function
0	00	—	0	Does nothing
1	01	A	1	Send next character to printer only
2	02	B	0	Enable printer
3	03	C	0	Disable printer
4	04	D	0	Write text at text cursor
5	05	E	0	Write text at graphics cursor
6	06	F	0	Enable VDU drivers
7	07	C	0	Make a short bleep (BEL)
8	08	H	0	Move cursor back one character
9	09	I	0	Move cursor forward one character
10	0A	J	0	Move cursor down one line
11	0B	K	0	Move cursor up one line
12	0C	L	0	Clear text area
13	0D	M	0	Carriage return
14	0E	N	0	Paged mode on
15	0F	O	0	Paged mode off
16	10	P	0	Clear graphics area
17	11	Q	1	Define text colour
18	12	R	2	Define graphics colour
19	13	S	5	Define logical colour
20	14	T	0	Restore default logical colours
21	15	U	0	Disable VDU drivers or delete current line
22	16	V	1	Select screen MODE
23	17	W	9	Re-program display character
24	18	X	8	Define graphics window
25	19	Y	5	PLOT K,X,Y
26	1A	Z	0	Restore default windows
27	1B	[	0	ESCAPE value
28	1C	\	4	Define text window
29	1D	]	4	Define graphics origin
30	1E	*	0	Home text cursor to top left of window
31	1F	_	2	Move text cursor to X,Y
127	7F	DEL	0	Backspace and delete

Appendix E - Plot Number Summary

0	Move relative to last point
1	Draw relative to last point in current foreground colour
2	Draw relative to last point in logical inverse colour
3	Draw relative to last point in current background colour
4	Move absolute
5	Draw absolute in current foreground colour
6	Draw absolute in logical inverse colour
7	Draw absolute in current background colour

Higher PLOT numbers have other effects which are related to the effects given by the values above.

8—15	Last point in line omitted when 'inverted' plotting used
16—23	Using a dotted line
24—31	Dotted line omitting last point
32—63	Reserved for Graphics Extension ROM
64—71	Single point plotting
72—79	Horizontal line filling
80—87	Plot and fill triangle
88—95	Horizontal line blanking (right only)
96—255	Reserved for future expansions

Horizontal line filling

These PLOT numbers start from the specified X,Y co-ordinates. The graphics cursor is then moved left until the first non-background pixel is encountered. The graphics cursor is then moved right until the first non-background coloured pixel is encountered on the right hand side. If the PLOT number is 73 or 77 then a line will be drawn between these two points in the current foreground colour. If the PLOT number is 72 or 76 then no line is drawn but the cursor movements are made (these may be read using OSWORD call with A=&D/13, see Chapter 15).

Horizontal line blanking (right)

These PLOT numbers can be used to 'undraw' an object on the screen. They have an the opposite effect to those of the horizontal line filling functions except that the graphics cursor is moved right only. PLOT numbers 91 and 95 will cause a line to be drawn from the specified co-ordinates to the nearest background coloured pixel to the right in the background colour. PLOT numbers 89 and 93 move the graphics cursor but do not cause the line to be blanked.

Appendix F - Screen mode layouts

The following pages list the eight modes that the BBC Model B is capable of displaying. A table of values that are programmed into the 6845 registers are given, along with a diagram showing the memory arrangement of the screen.

NOTE: The screen layout of all screen modes is only valid after a clear screen, where no scrolling has occurred.

MODE 0 Screen layout

Graphics 640 x 256

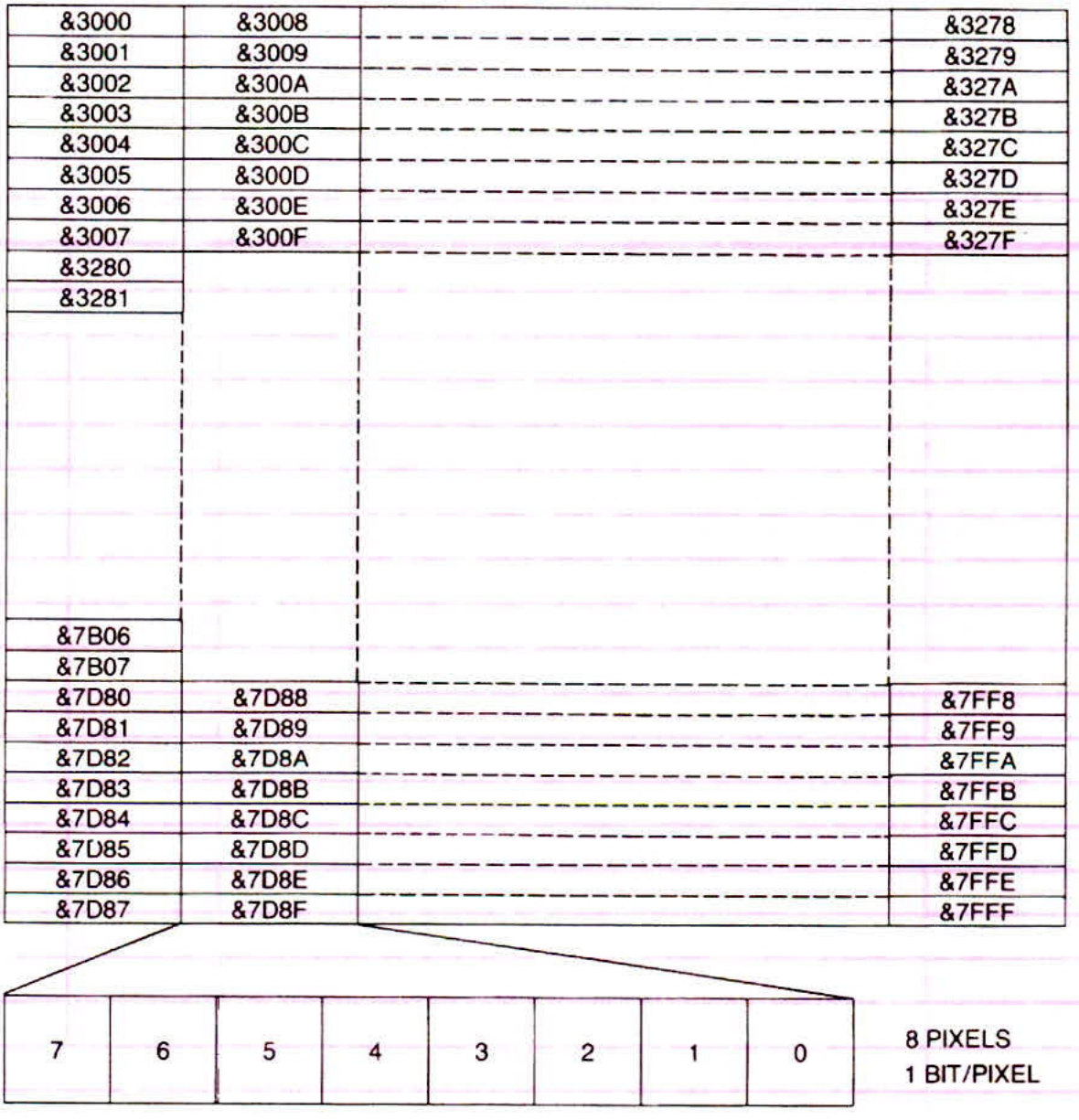
Colours 2

Text 80 x 32

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&7F	127
R1	Characters per line	&50	80
R2	Horizontal sync position	&62	98
R3	Horizontal sync width	&08	8
	Vertical sync time	&02	2
R4	Vertical total	&26	38
R5	Vertical total adjust	&00	0
R6	Vertical displayed characters	&20	32
R7	Vertical sync position	&22	34
R8	Interlace mode bits 0,1	&01	1
	Display delay bits 4,5	&00	0
	Cursor delay bits 6,7	&00	0
R9	Scan lines per character	&07	7
R10	Cursor start blink type	&67	103
	Cursor start (bits 0-4)	&07	7
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&08	8
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 0



N.B. The screen layout is only as shown after a clear screen, it will change when the screen is hard scrolled.

MODE 1 Screen layout

Graphics 320 x 256

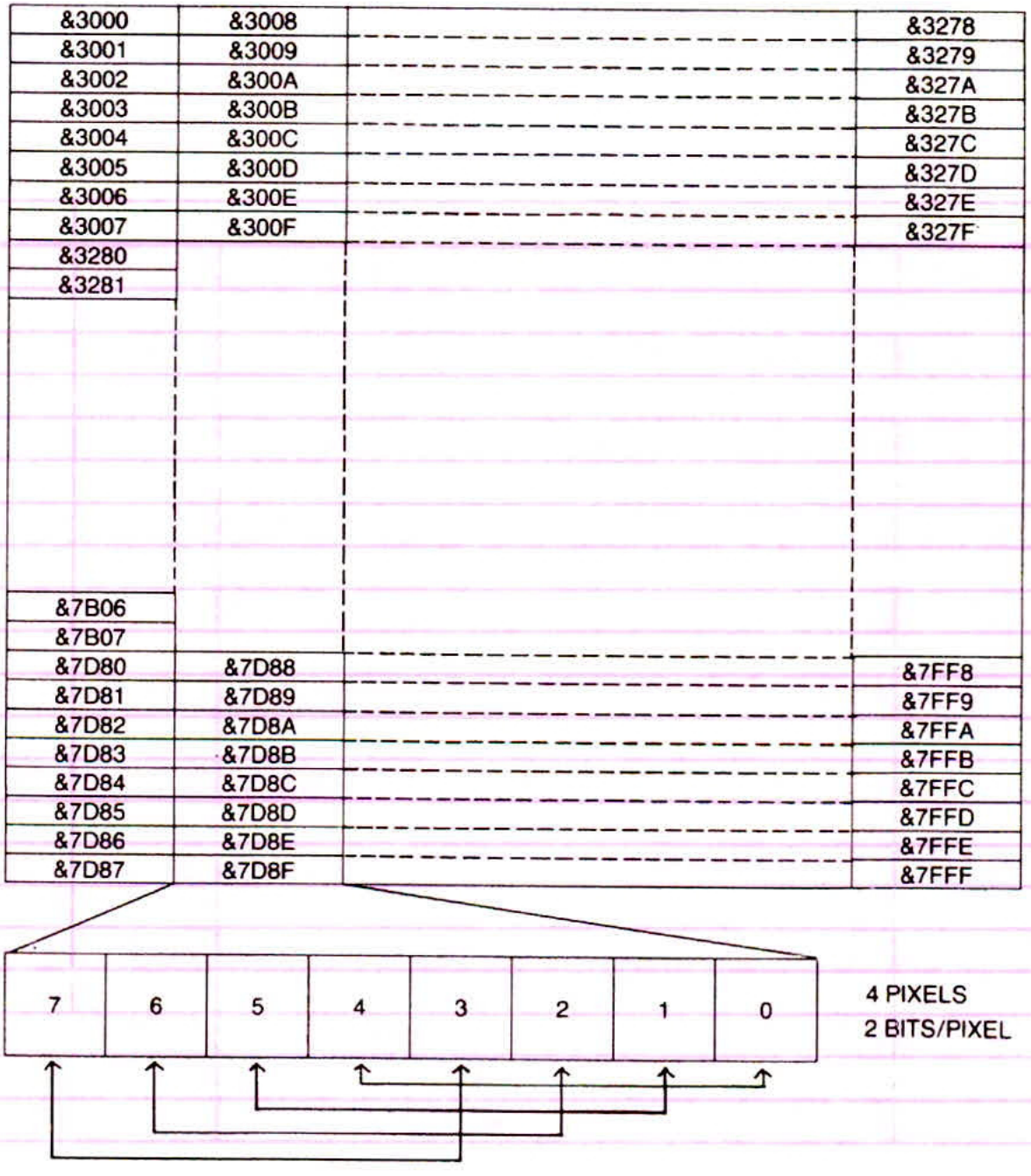
Colours 4

Text 40 x 32

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&7F	127
R1	Characters per line	&50	80
R2	Horizontal sync position	&62	98
R3	Horizontal sync width	&08	8
	Vertical sync time	&02	2
R4	Vertical total	&26	38
R5	Vertical total adjust	&00	0
R6	Vertical displayed characters	&20	32
R7	Vertical sync position	&22	34
R8	Interlace mode bits 0,1	&01	1
	Display delay bits 4,5	&00	0
	Cursor delay bits 6,7	&00	0
R9	Scan lines per character	&07	7
R10	Cursor start blink type	&67	103
	Cursor start (bits 0-4)	&07	7
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&08	8
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 1



N.B. The screen layout is only as shown after a clear screen, it will change when the screen is hard scrolled.

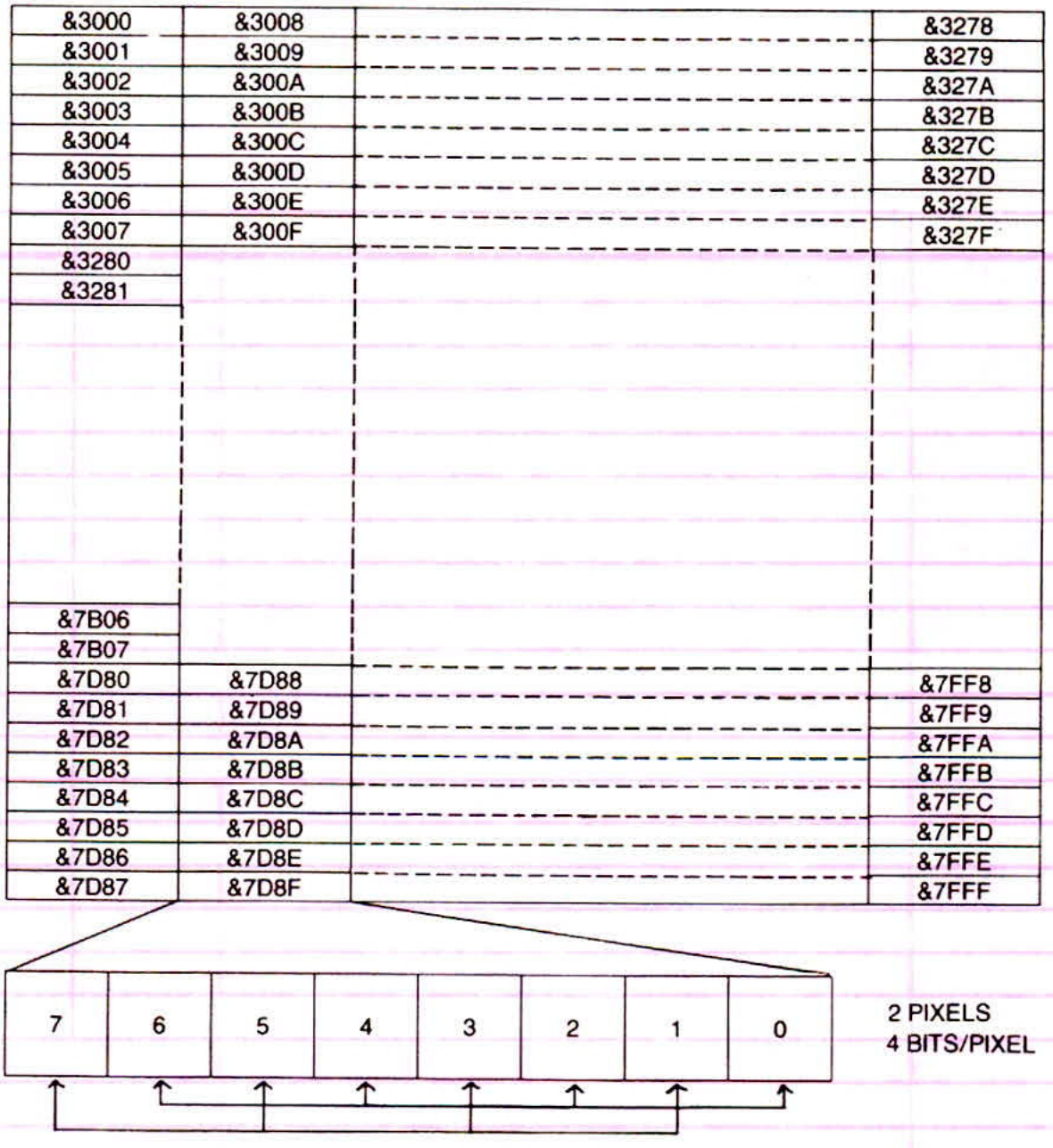
MODE 2 Screen layout

Graphics	160 x 256
Colours	16
Text	20 x 32

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&7F	127
R1	Characters per line	&50	80
R2	Horizontal sync position	&62	98
R3	Horizontal sync width	&08	8
	Vertical sync time	&02	2
R4	Vertical total	&26	38
R5	Vertical total adjust	&00	0
R6	Vertical displayed characters	&20	32
R7	Vertical sync position	&22	34
R8	Interlace mode bits 0,1	&01	1
	Display delay bits 4,5	&00	0
	Cursor delay bits 6,7	&00	0
R9	Scan lines per character	&07	7
R10	Cursor start blink type	&67	103
	Cursor start (bits 0-4)	&07	7
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&08	8
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 2


&3000
&3001
&3002
&3003
&3004
&3005
&3006
&3007
&3008
&3009
&300A
&300B
&300C
&300D
&300E
&300F
&3278
&3279
&327A
&327B
&327C
&327D
&327E
&327F
&3280
&3281
&7B06
&7B07
&7D80
&7D81
&7D82
&7D83
&7D84
&7D85
&7D86
&7D87
&7D88
&7D89
&7D8A
&7D8B
&7D8C
&7D8D
&7D8E
&7D8F
&7FF8
&7FF9
&7FFA
&7FFB
&7FFC
&7FFD
&7FFE
&7FFF
7
6
5
4
3
2
1
0
2 PIXELS
4 BITS/PIXEL

MODE 3 Screen layout

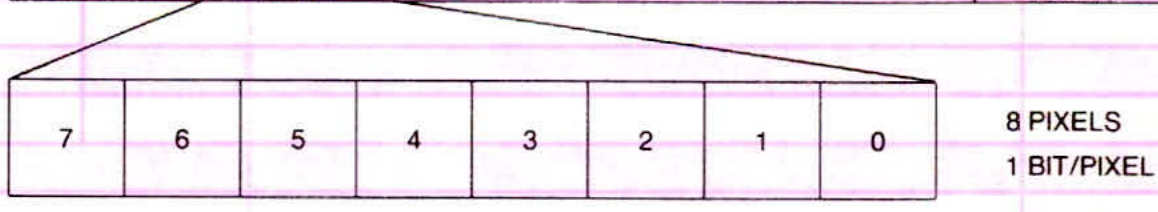
Graphics	Not available
Colours	2
Text	80 x 25

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&7F	127
R1	Characters per line	&50	80
R2	Horizontal sync position	&62	98
R3	Horizontal sync width	&08	8
	Vertical sync time	&02	2
R4	Vertical total	&1E	30
R5	Vertical total adjust	&02	2
R6	Vertical displayed characters	&19	25
R7	Vertical sync position	&1B	27
R8	Interlace mode bits 0,1	&01	1
	Display delay bits 4,5	&00	0
	Cursor delay bits 6,7	&00	0
R9	Scan lines per character	&09	9
R10	Cursor start blink type	&67	103
	Cursor start (bits 0-4)	&07	7
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&09	9
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 3

&4000	&4008		&4278
&4001	&4009		&4279
&4002	&400A		&427A
&4003	&400B		&427B
&4004	&400C		&427C
&4005	&400D		&427D
&4006	&400E		&427E
&4007	&400F		&427F
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK
&4280			
&4281			
&7980			
BLANK			
BLANK			
&7C00	&7C08		&7E78
&7C01	&7C09		&7E79
&7C02	&7C0A		&7E7A
&7C03	&7C0B		&7E7B
&7C04	&7C0C		&7E7C
&7C05	&7C0D		&7E7D
&7C06	&7C0E		&7E7E
&7C07	&7C0F		&7E7F
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK



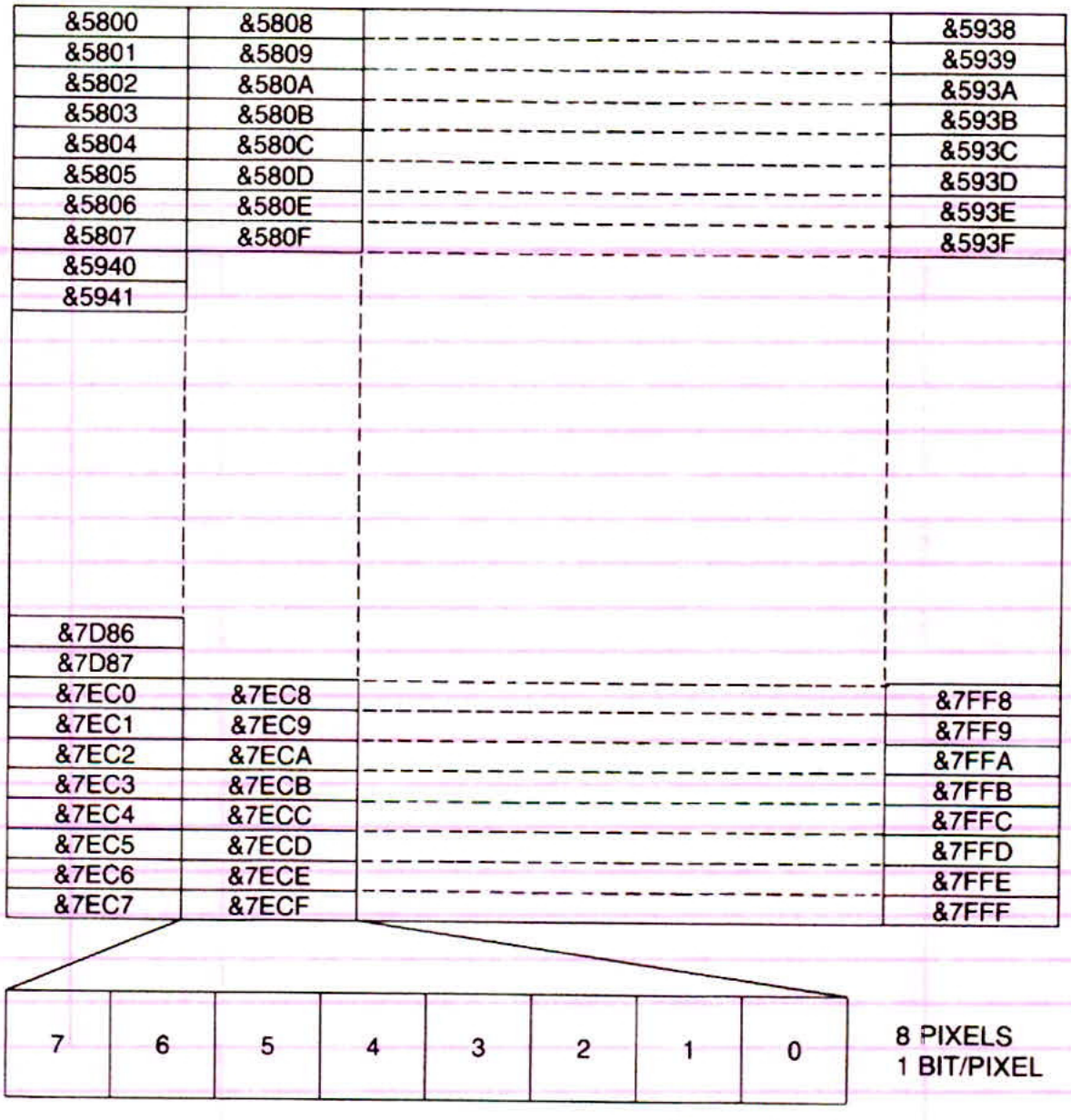
MODE 4 Screen layout

Graphics 320 x 256
Colours 2
Text 40 x 32

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&3F	63
R1	Characters per line	&28	40
R2	Horizontal sync position	&31	49
R3	Horizontal sync width	&04	4
	Vertical sync time	&02	2
R4	Vertical total	&26	38
R5	Vertical total adjust	&00	0
R6	Vertical displayed characters	&20	32
R7	Vertical sync position	&22	34
R8	Interlace mode bits 0,1	&01	1
	Display delay bits 4,5	&00	0
	Cursor delay bits 6,7	&00	0
R9	Scan lines per character	&07	7
R10	Cursor start blink type	&67	103
	Cursor start (bits 0-4)	&07	7
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&08	8
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 4



N.B. The screen layout is only as shown after a CLS, it will change when the screen is hard scrolled. &4000 should be subtracted for each screen location on a model A.

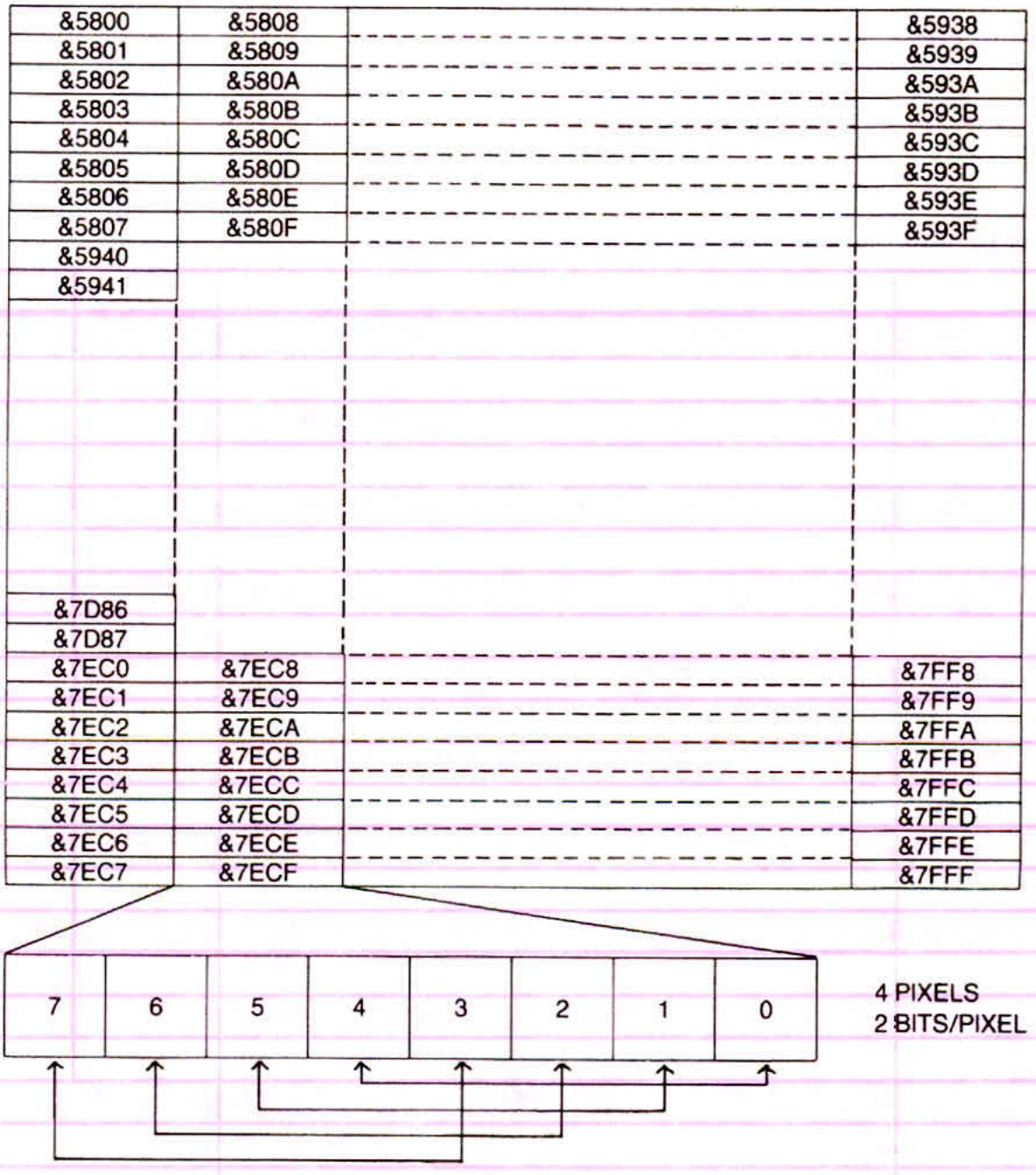
MODE 5 Screen layout

Graphics 160 x 256
Colours 4
Text 20 x 32

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&3F	63
R1	Characters per line	&28	40
R2	Horizontal sync position	&31	49
R3	Horizontal sync width	&04	4
	Vertical sync time	&02	2
R4	Vertical total	&26	38
R5	Vertical total adjust	&00	0
R6	Vertical displayed characters	&20	32
R7	Vertical sync position	&22	34
R8	Interlace mode bits 0,1	&01	1
	Display delay bits 4,5	&00	0
	Cursor delay bits 6,7	&00	0
R9	Scan lines per character	&07	7
R10	Cursor start blink type	&67	103
	Cursor start (bits 0-4)	&07	7
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&08	8
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 5



N.B. The screen layout is only as shown after a CLS, it will change when the screen is hard scrolled. &4000 should be subtracted for each screen location on a model A.

MODE 6 Screen layout

Graphics Not available

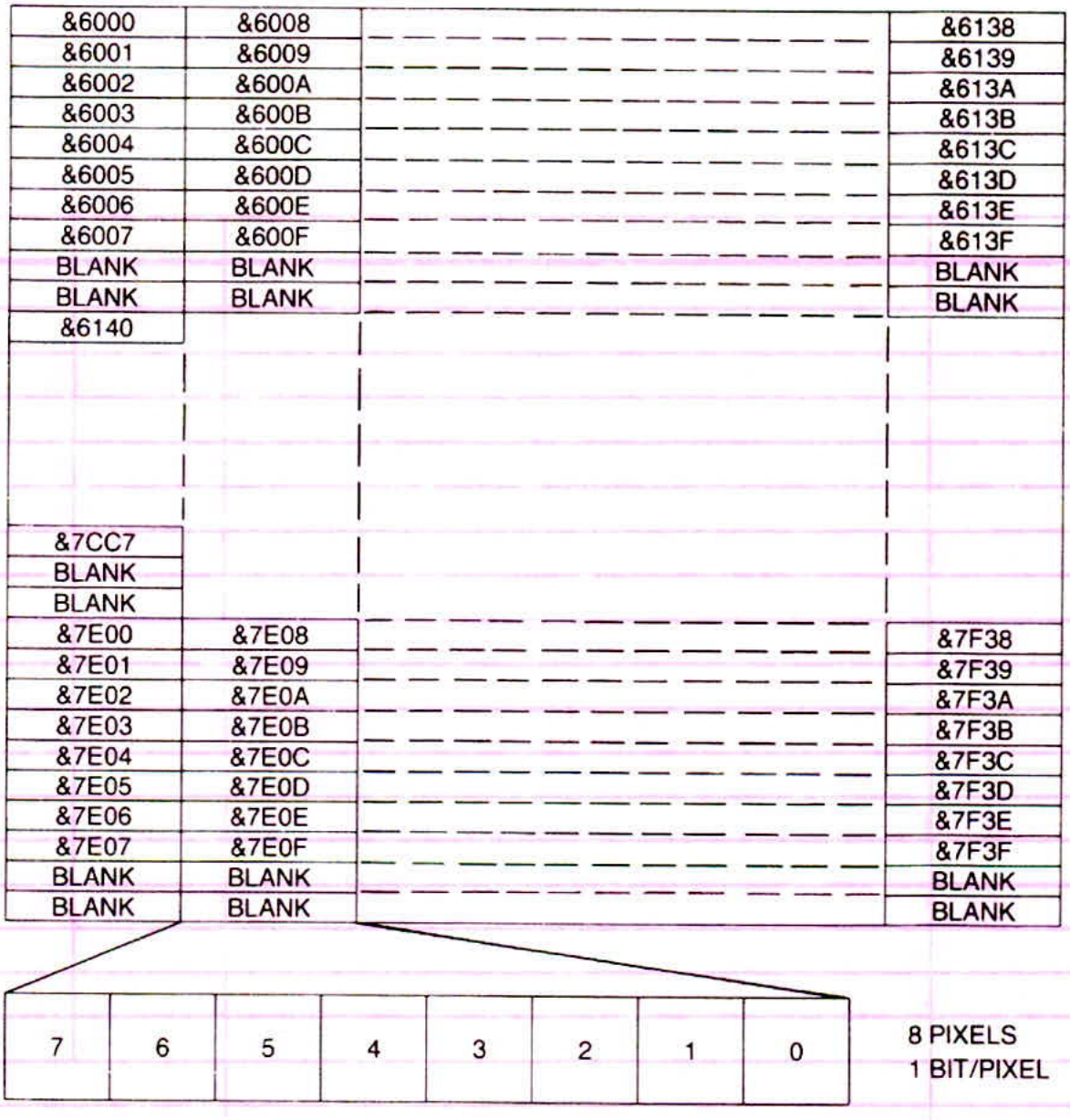
Colours 2

Text 40 x 25

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&3F	63
R1	Characters per line	&28	40
R2	Horizontal sync position	&31	49
R3	Horizontal sync width	&04	4
	Vertical sync time	&02	2
R4	Vertical total	&1E	30
R5	Vertical total adjust	&02	2
R6	Vertical displayed characters	&19	25
R7	Vertical sync position	&1B	27
R8	Interlace mode bits 0,1	&01	1
	Display delay bits 4,5	&00	0
	Cursor delay bits 6,7	&00	0
R9	Scan lines per character	&09	9
R10	Cursor start blink type	&67	103
	Cursor start (bits 0-4)	&07	7
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&09	9
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 6



N.B. The screen layout is only as shown after a CLS, it will change when the screen is hard scrolled. &4000 should be subtracted for each location on a model A.

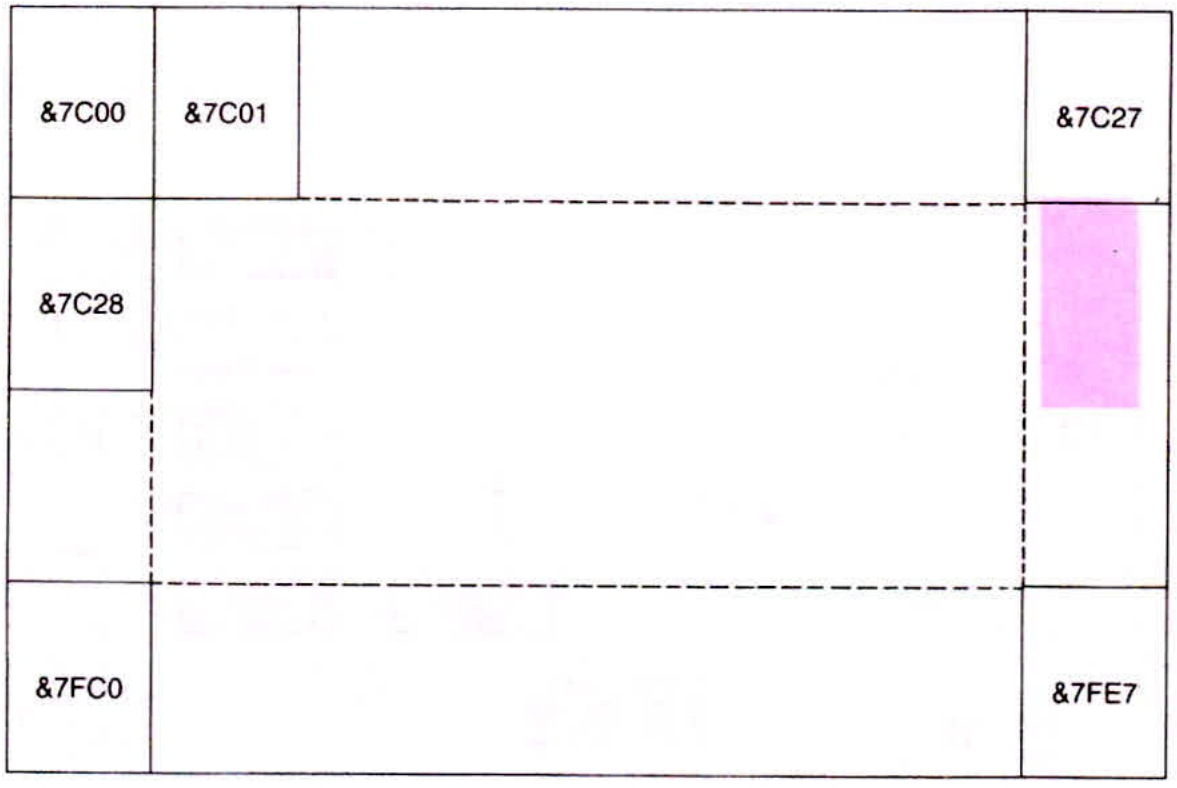
MODE 7 Screen layout

Graphics	TELETEXT graphics only
Colours	TELETEXT
Text	40 x 25

Registers

Register	Purpose	Hex	Dec
R0	Horizontal total	&3F	63
R1	Characters per line	&28	40
R2	Horizontal sync position	&33	51
R3	Horizontal sync width	&04	4
	Vertical sync time	&02	2
R4	Vertical total	&1E	30
R5	Vertical total adjust	&02	2
R6	Vertical displayed characters	&19	25
R7	Vertical sync position	&1B	27
R8	Interlace mode bits 0,1	&03	3
	Display delay bits 4,5	&01	1
	Cursor delay bits 6,7	&02	2
R9	Scan lines per character	&12	18
R10	Cursor start blink type	&72	114
	Cursor start (bits 0-4)	&12	18
	Cursor blink (bit 6)	&01	1
	Cursor type (bit 5)	&01	1
R11	Cursor end	&13	19
R12,R13	Screen start address	Variable	
R14,R15	Cursor position	Variable	
R16,R17	Light pen position	Variable	

MODE 7



N.B. The screen layout is only as shown after a CLS, it will change when the screen is hard scrolled. &4000 should be subtracted for each location on a model A.

Appendix G - Link options

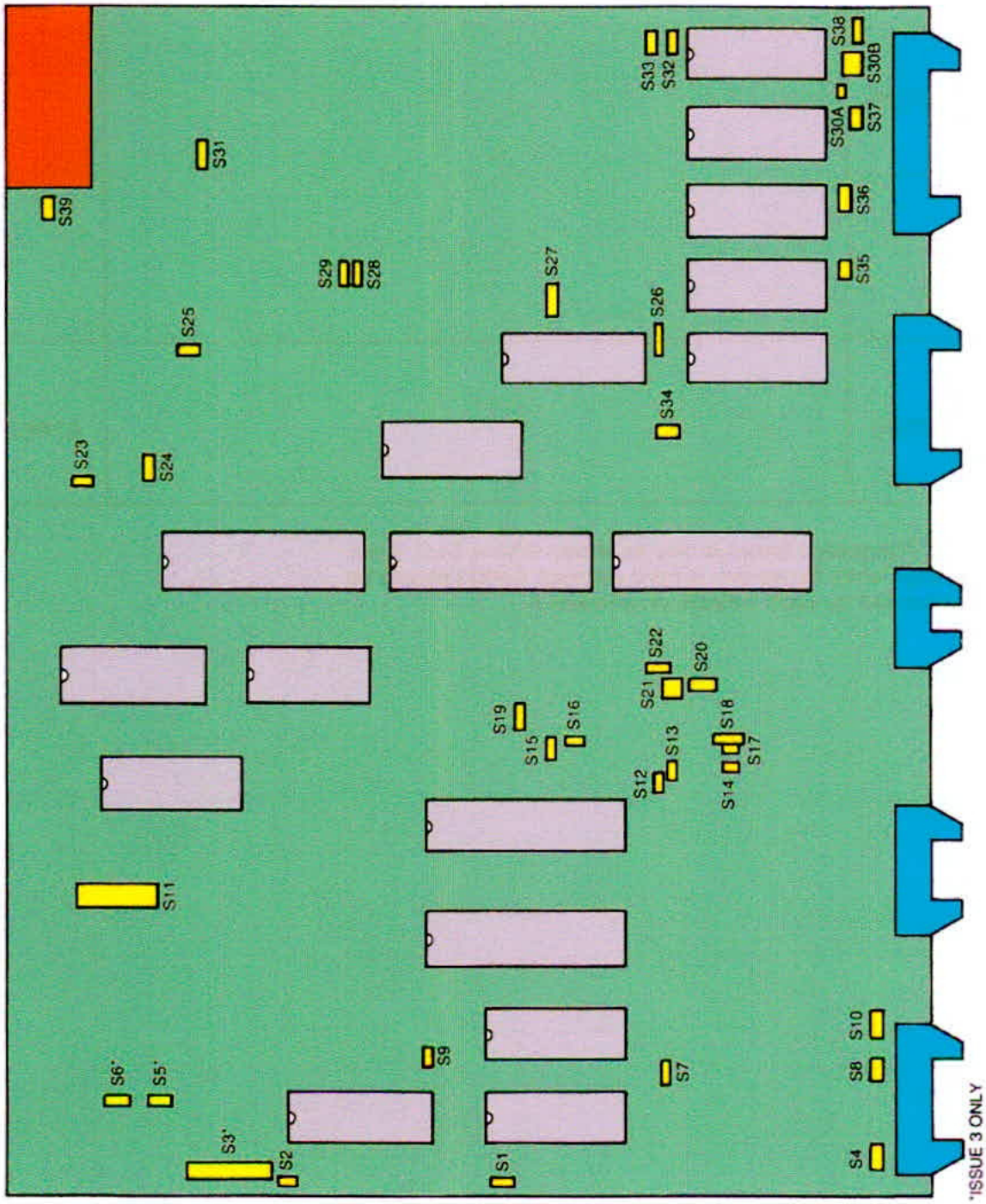


Figure A.1 – Link positions on the Model B Circuit board

There are several options which can be selected using the links on the main circuit board. These generally take one of three forms, a printed circuit track, a soldered wire link or a plug-in jumper. In the list which follows, the link positions are described in points of the compass. On this basis NORTH is the direction looking from the keyboard towards the back of the BBC microcomputer. All of the link positions are illustrated in Figure A.1.

1.	NORTH	selects printer strobe on printer connector
	SOUTH	selects 6522 CA2 connected to printer strobe
	TYPE	circuit board track
		NOTE: Not fitted to issues 1-3.
2.	OPEN	enables Econet NMI
	CLOSED	disables Econet NMI
	TYPE	wire link
		NOTE: ***NEVER*** fit this link with 1C91 fitted.
3.		Selects the clock base frequency for Econet. This bank of 9 links is only fitted on issues 1-3.
4.	EAST	5¼ inch DISC select
	WEST	8 inch DISC select
	TYPE	circuit board track
5.	NORTH	enable Econet clock
	SOUTH	disable Econet clock
		NOTE: Only fitted to issues 1-3.
6.	NORTH	divide Econet clock by 2
	SOUTH	divide Econet clock by 4
		NOTE: Only fitted to issues 1-3.
7.	WEST	connects pin 30 of 8271 FDC chip to +5 volts
	EAST	connects pin 30 of 8271 FDC chip to 0 volts
8.	OPEN	disconnects disc head load signal from PL8
	CLOSED	connects disc head load signal to PL8

9.

OPEN	enables disc NMI
CLOSED	disables Disc NMI

NOTE: This link must be OPEN when IC78 is fitted.

10.

WEST	select 5¼ inch disc
EAST	select 8 inch disc

11. This block of 8 links selects the Econet ID. The NORTH connection is the least significant bit. A different Econet ID will be assigned to each station on an Econet system. Refer to the Econet manual for more details.

12.

CLOSED	connects ROM select line A to 0v (model A)
OPEN	ROM select line is driven by IC76 (model B)

NOTE: Do NOT make this link with IC76 fitted.

13.

CLOSED	connects ROM select line B to 0v (model A)
OPEN	ROM select line is driven by IC76 (model B)

NOTE: Do NOT make this link with IC76 fitted.

14.

OPEN	enables JIM, disables ROM output from page &FD
CLOSED	disables JIM, enables ROM output from page &FD

NOTE: Link 16 must be CLOSED if link 14 is OPEN.

15.

OPEN	enables fast access to page &FD
CLOSED	disables fast access to page &FD via IC23

NOTE: Link 15 must be CLOSED when link 17 is OPEN.

16.

OPEN	enable fast access to page &FC via IC23
CLOSED	disable fast access to page &FC

NOTE: Link 16 must be CLOSED if link 14 is OPEN.

17.

OPEN	disable FRED, enable ROM output from page &FC
CLOSED	enable FRED, disable ROM output from page &FC

NOTE: Link 15 must be CLOSED if link 17 is OPEN.

18.	NORTH	fast access to IC100 memory chip
	SOUTH	slow access to IC100 memory chip
19.	WEST	slow access to memory chips IC52, IC88 and IC101
	EAST	fast access to memory chips IC52, IC88 and IC101
		NOTE: Diodes D10, D11 and D12 can be removed to speed up ROMs IC101, IC88 and IC52 respectively when link 19 is in the WEST position.
20.	NORTH	ROMSEL provides HIGH ROM select bit to IC20
	SOUTH	A13 provides HIGH ROM select bit to IC20
21.	NORTH/SOUTH	IC51 = blocks &08-&0B
	x2	IC52, IC88, IC100 & IC101 = blocks &0C-&0F
	EAST/WEST	IC5I = blocks &0C-&0F
	x2	IC52, IC88, IC100 & IC101 = blocks &08-&0B
22.	NORTH	ROMSEL provides the LOW ROM select bit to IC20
	SOUTH	A12 provides the LOW ROM select bit to IC20
23.	OPEN	RS423 DATA line not terminated
	CLOSED	RS423 DATA line is terminated
		NOTE: See Note for link 24.
24.	OPEN	RS423 CTS line not terminated
	CLOSED	RS423 CTS line is terminated
		NOTE: It is not normally necessary to terminate the RS423 lines unless high baud rates or long interconnecting wires are being used.
25.	NORTH	32K RAM select (model B)
	SOUTH	16K RAM select (model A)

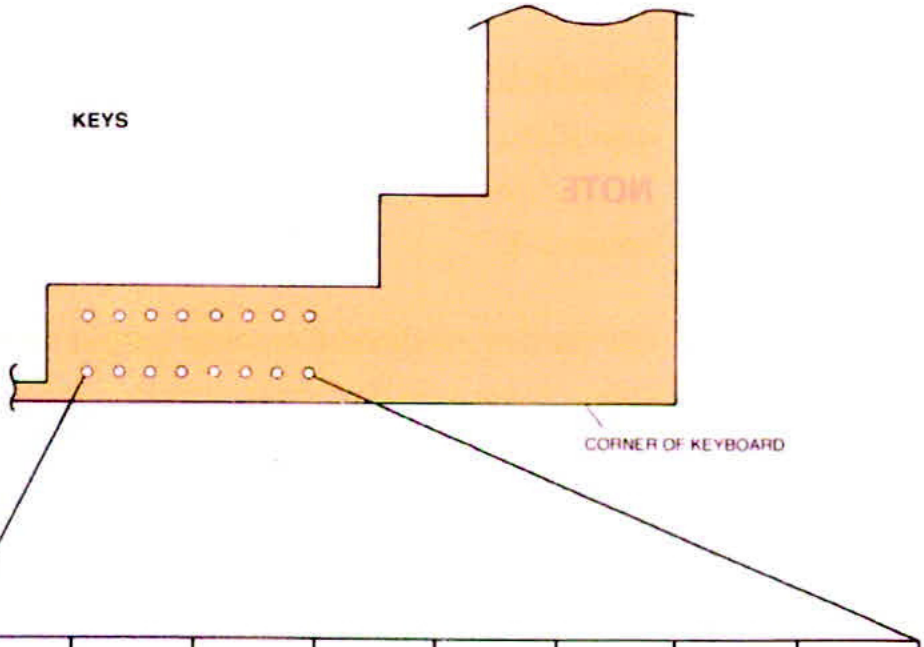
26.	WEST	NORMAL video output
	EAST	INVERTED video output
		NOTE: If fitted with a VC2023 Video ULA, no link must be fitted here. A patch wire between the centre pin and Pin 1 of IC10 will be fitted to correct a fault in the ULA.
27.	WEST	5¼ inch disc 8MHz clock select
	EAST	8 inch disc 16MHz clock select
28.	WEST	base baud rate select
	EAST	1200 baud rate select
		NOTE: RS423 rate is affected with link 28 set EAST.
29.	WEST	1200 baud rate select
	EAST	base baud rate select
		NOTE: RS423 rate is affected with link 29 set WEST.
30a &		used to 'WIRE-OR' two or more ROM select signals
30b		**NOTE**: See links 34-38 and the circuit diagram.
31.	WEST	positive CSYNC to RGB video output
	EAST	negative CSYNC to RGB video output
32.	WEST	connects A13 to A13 pin of IC52 and IC88
	EAST	connects +5 volts to A13 pin of IC52 and IC88
33.	WEST	connects A13 to A13 pin of IC100 and IC101
	EAST	connects +5 volts to A13 pin of IC100 and IC101
34.	OPEN	allows the OS ROM to be 'WIRE-OR'ed
	CLOSED	use ordinary OS ROM select from IC20
		NOTE: See full circuit diagram for more details.

35.	OPEN	allows IC52 ROM to be 'WIRE-OR'ed
	CLOSED	use ordinary IC52 ROM select from IC20
		NOTE: Link only available on issue 4. Replaces R125 in issues 1-3.
36.	OPEN	allows IC88 ROM to be 'WIRE-OR'ed
	CLOSED	use ordinary IC88 ROM select from IC20
		NOTE: Link only available on issue 4. Replaces R142 in issues 1-3.
37.	OPEN	allows IC100 ROM to be 'WIRE-OR'ed
	CLOSED	uses IC100 ROM select from IC20
		NOTE: Link only available on issue 4. Replaces R149 in issues 1-3.
38.	OPEN	allows IC101 ROM to be 'WIRE-OR'ed
	CLOSED	uses IC101 ROM select from IC20
		NOTE: Link only available on issue 4. Replaces R153 in issues 1-3.
39.	OPEN	standard monochrome monitor output on BNC socket
	CLOSED	colour output on BNC video socket

Appendix H - The keyboard circuit

The keyboard circuit is connected to the main printed circuit board by two ribbon cables. The larger of these is fitted to all BBC microcomputers and carries all key pressed and start-up option data. The smaller one is connected directly to the speech processor. With the speech system installed, this extra connector allows serial ROMs to be plugged in to the hole on the left hand side of the keyboard.

In the lower right hand corner of the keyboard printed circuit board, there are two rows of eight holes. These are the keyboard link options. The operation of these links is defined by a made or unmade connection. The function of each link bit is described under OSBYTE &FF. The diagram below illustrates the layout of the links. If a user wishes to vary the settings fairly often, it is a good idea to buy a standard 8 way SPST switch in a 16 pin DIL package, and solder it onto the keyboard.



BIT NUMBER	7	6	5	4	3	2	1	0
SWITCH NUMBER	1	2	3	4	5	6	7	8
DESCRIPTION	NOT USED	NOT USED	DISC DRIVE SPEED SELECT		DEFINES ACTION OF SHIFT + BREAK	SCREEN MODE AT POWER UP AND AFTER RESET		

Figure A.2 – Illustration of keyboard links

NOTE: On some later keyboard types, the socket can be in a vertical orientation. In this case, switch 1 is at the top.

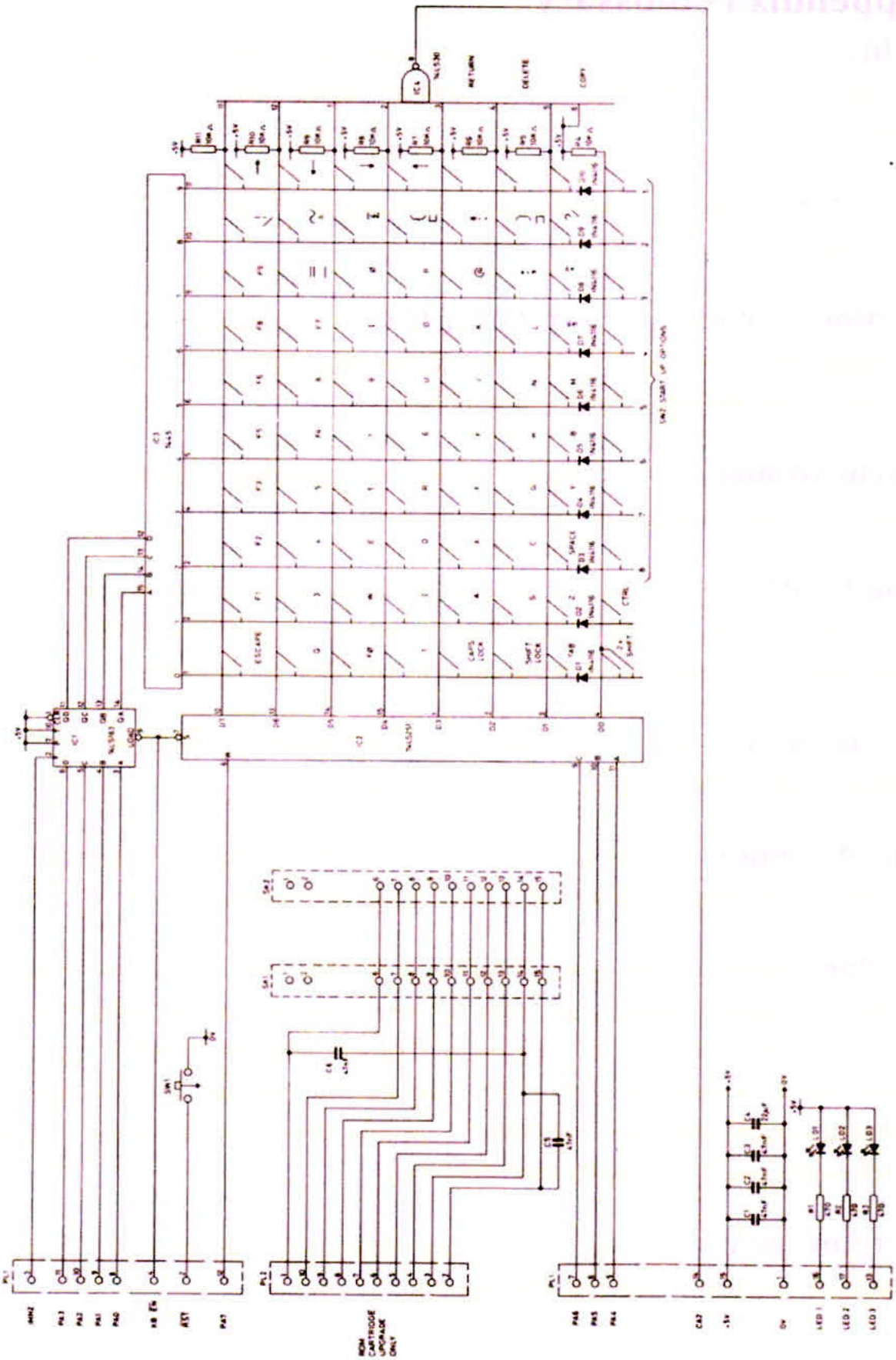


Figure A.3 – Complete keyboard circuit diagram

Appendix I - Glossary

Address Bus - a set of 16 connections, each one of which can be set to logic 0 or logic 1. This allows the CPU to address &FFFF (65536) different memory locations.

Active low - signals which are 'active low' are said to be valid when they are at logic level 0.

Analogue to digital converter (ADC) - this is a chip which can accept an analogue voltage at one of its inputs and provide a digital output of that voltage. The ADC in the BBC microcomputer is a 7002.

Asynchronous - two devices which are operating independently of one another are said to be operating asynchronously.

Baud Rate - used to define the speed at which a serial data link transfers data. One baud is equal to 1 bit of data transferred per second. The standard cassette baud rate of 1200 baud is therefore equal to 1200 bits per second.

Bidirectional - a communication line is bidirectional if data can be sent and received over it. The data bus lines are bidirectional.

Bit of memory - this is the fundamental unit of a computer's memory. It may only be in one of two possible states, usually represented by a 0 or 1.

Buffer - there are two types of buffer in the BBC microcomputer. A software buffer is an area of memory set aside for data in the process of being transferred from one device or piece of software to another. A hardware buffer is put into a signal line to increase the line's drive capability. For example, the printer port has buffered outputs which are capable of supplying several milliamperes. The inputs to the buffer could only supply microamperes.

Byte of memory - 8 bits of memory. Data is normally transferred between devices one byte at a time over the data bus.

Chip - derived from the small piece of silicon wafer or chip which has all of the computer logic circuits etched into it. A chip is normally packaged in a black plastic case with small metal leads to connect it to the outside world.

Clock - since there are so many devices in the BBC microcomputer, it is necessary to provide some master timing reference to which all data transfers are tied. The clock provides this synchronisation. A 2MHz clock is applied to the CPU, but this can be stretched into a 1MHz clock when slow peripherals such as the 6522s are being accessed. See the section on the 1MHz bus in Chapter 23 for more details about cycle stretching.

CPU (Central processing unit) - the 6502A in the BBC microcomputer. It is this chip which does all of the computing work associated with running programs. A Model B+ Uses a 6512 CPU, which is similar, but a later version. It uses the same instruction set.

Cycle - this is usually applied to the system clock. A complete clock cycle is the period between a clock going high, low, then high again. See 'clock'.

Data bus - a set of eight connections over which all data transactions between devices in the BBC microcomputer take place.

Field - a space allocated for some data in a register, or in a program listing. For example, in an Assembly language program, the first few spaces are allocated to the line number field, the next few spaces are allocated to the label field, and so on.

FRAM - Stands for Ferroelectric Random Access Memory. A type of Static RAM that retains its contents, even with the power removed, without the need for a battery backup.

Handshaking - this type of communications protocol is used when data is being transferred between two asynchronous devices. Two handshaking lines are normally required. One of these is a 'data ready' signal from the originating device to the receiving device. When the receiving device has accepted the data, it sends a 'data taken' signal back to the originating device, which then knows that it can send the second lot of data and so on.

High - sometimes used to designate logic '1'

Interrupt - this signal is produced by peripheral devices and is always directed to the 6502A CPU. Upon receiving an interrupt, the 6502 will normally run a special interrupt routine program before continuing with the task in hand before it was interrupted.

Latch - a latch is used to retain information applied to it after the data has been removed. It is rather like a memory location except that the outputs from the bits within the latch are connected to some hardware.

LED (Light emitting diode) - acts like a diode by only allowing current to pass in one direction. Light is emitted whilst current is passed.

Low - sometimes used to designate logic '0'.

Machine code - the programs produced by the 6502 **BASIC** Assembler are machine code. A machine code program consists of a series of bytes in memory which the 6502 can execute directly.

Mnemonic - the name given to the text string which defines a particular 6502 operation in the **BASIC** assembler. `LDA` is a mnemonic which means 'load accumulator'.

Opcode - the name given to the binary code of a 6502 instruction. For example, &AD is the opcode which means 'load accumulator'.

Open Collector - this is a characteristic of a transistor output line. It simply means that the collector pin of the transistor is not driving a resistor load, i.e. it is 'open'.

Operand - a piece of data on which some operation is performed. Usually the operand will be a byte in the accumulator of the 6502, or a byte in some memory location.

Page - a page of memory in the 6502 memory map is &100 (256) bytes long. There are therefore 256 pages in the entire address space. 256 pages of 256 bytes each account for the 65536 bytes of addressable memory.

Parallel - parallel data transfers occur when data is sent along two or more lines at once. The system data bus for example has eight lines operating in parallel.

Peripheral - any device connected to the 6502 central processor unit, such as the analogue port, printer port, econet and disc interface etc., but not including the memory.

Poll - most of the hardware devices on the BBC microcomputer generate interrupts to the 6502 CPU. If interrupts have been enabled, the CPU has to find out which device generated the interrupt. It does this by successively reading status bytes from each of the hardware devices which could have caused an interrupt. This successive reading of devices is called 'polling'.

RAM (Random access memory) - the main memory in the BBC microcomputer is RAM because it can be both written to and read from.

Refresh - all of the memories in the standard BBC microcomputer are dynamic memories. This means that they have to be refreshed every few milliseconds so that their data is not lost. The refreshing function is performed by the 6845 as it accesses memory regularly for video output. Static RAM is used in some Sideways RAM expansions, this type of RAM does not need a refresh cycle to retain it's memory.

Register - the 6502 and many of the peripheral devices in the BBC microcomputer contain registers. These are effectively one byte memory locations which do not necessarily reside in the main memory map. All software on the 6502 makes extensive use of the internal registers for programming. The bits in most peripheral registers define the operation of a particular piece of hardware, or tell the processor something about that peripheral's state.

Rollover - this is a function provided on the keyboard to cope with fast typists. Two keys can be pressed at once. The previous key with a finger being removed, and the next key with the finger hitting the key. The software in the operating system ensures that rollover normally operates correctly. It doesn't operate at all when the *shift* key is held down.

ROM (Read only memory) - as the name implies, ROM can only be read from and cannot be modified by being written to. The MOS and **BASIC** plus any resident software in the BBC microcomputer are held in ROMs.

Serial - data transmitted along only one line is transmitted serially. Serial data transmission is normally slower than parallel data transmission, because only one bit instead of several bits are transferred at a time. This is not necessarily the case with modern computers and processors nowadays.

Stack - a page of memory in the 6502 used for temporary storage of data. Data is pushed onto a stack in sequence, then removed by pulling the data off the stack. The last byte to be pushed is the first byte to be pulled off again. The stack is used to store return addresses from subroutines. Page &01 is used for the stack in the BBC microcomputer.

Static RAM – A RAM that does not require a refresh cycle to retain its data. Generally used for Sideways RAM expansions. Like Dynamic RAM, it loses its data when power is lost.

Transducer - a device which converts some analogue quantity such as temperature, humidity or gas concentration into another quantity, usually voltage or current, which can be measured by a computer.

Tristate - in the BBC microcomputer, it is often necessary to connect the outputs of several chips together. At any one time only one of the chips should have priority. If the other chips were allowed to be in the opposite logic state, the power supplies would effectively be shorted through the chips. A special tristate level is therefore provided on some chips, to allow them to turn off their outputs and share the data bus.

ULA (Uncommitted logic array) - these are special chips which contain a large number of logic gates. The connection between the gates is defined when the chip is manufactured. Acorn have produced several special ULAs, two of which are used in the BBC Micro - one containing most of the serial circuitry, the other containing some of the video circuitry. They are an early predecessor to the CPLD or FPGA chips we would use nowadays, but are not reprogrammable.

Appendix J - The shadow screen

This section only applies to the BBC Microcomputer model B+. A standard BBC Model B does not have this facility, unless upgraded by a 3rd party hardware board, such as the Watford Electronics 32K RAM card.

On a Model B+, there are two areas of memory in the computer that can be selected for use as display memory: from &3000 (variable, according to the display **MODE** selected) to &8000 in the main memory map, or an area outside the main memory map known as the 'shadow RAM'. Using the shadow RAM leaves the whole of user memory (i.e. from **PAGE** to &8000) free for your use.

Display **MODEs** **128** to **135** use the shadow RAM as the display memory by default, and are the shadow screen equivalents of **MODE**s **0** to **7** (respectively). To enter a shadow display **MODE**, simply type in its **MODE** number – for example:

```
MODE 135 <RETURN>
```

would enter the shadow screen equivalent of **MODE** **7**. Non-shadow mode may be re-entered by changing to a non-shadow **MODE**. The following command sequence illustrates the memory map changes which take place when switching between shadow and non-shadow **MODE**s:

```
MODE 3
P. ~HIMEM
     4000

MODE 131
P. ~HIMEM
     8000

MODE 3
P. ~HIMEM
     4000
```

(The screen would be cleared by each **MODE** change).

To force the operating system to enter shadow mode on subsequent **MODE** changes, (even if a non-shadow **MODE** is selected) type:

```
*SHADOW <RETURN>
```

To return to non-shadow mode type

```
*SHADOW 1 <RETURN>
```

followed by a **MODE** change to a non-shadow **MODE**. The following command sequence illustrates the memory map changes which take place when using the ***SHADOW** commands:

```
MODE 3
P. ~HIMEM
      4000
*SHADOW
P. ~HIMEM
      4000
MODE 3

P. ~HIMEM
      8000
```

MODE 131 would have the same effect as the second **MODE 3** command above, although in that case there would be no need for the preceding ***SHADOW** command. Continuing from the above sequence we have:

```
*SHADOW 1
P. ~HIMEM
8000
MODE 3

P. ~HIMEM
4000
```

Substituting **MODE 131** for **MODE 3** above would give:

```
P. ~HIMEM
8000
```

The ***SHADOW** commands are useful when switching between software which uses shadow RAM and software which doesn't.

Other shadow mode-related commands

***FX114** produces the same effect as ***SHADOW**, and ***FX114,1** produces the same effect as ***SHADOW 1**.

VDU22,<128+n> (used when in non-shadow mode) will select **MODE** n; shadow mode will not be entered and **HIMEM** will not be reset.

Shadow mode is retained across a soft break, but is reset to non-shadow by a hard break.

Appendix K - RetroClinic upgrades

If your machine was supplied as a refurbished unit by RetroClinic, it may have been pre-fitted with several upgrades. You will have received instructions pertaining to each of those add-ons when you bought your machine, but a brief summary of a few of the upgrades available follows here for reference.

Compact Flash Hard Drive

The main component of this upgrade is the BBC/IDE interface. It is a single chip custom programmable logic design, made to be compatible with various IDE patched operating systems for the BBC Micro and Master Series Computers. It is also designed for simplicity, ease of use, and above all, reliability.

Connections on the interface are very straightforward:

- 1MHz BUS – This will be fitted connected to the 34 way 1MHz bus connector underneath your computer, with the cable routing going underneath the PCB.
- Compact Flash socket – where the CF card plugs in!
- PWR – ***USE THE POWER CONNECTOR ON THE IDE INTERFACE, NOT THE ONE NEAR THE CF CARD!*** This is pin compatible with the standard 3.5" floppy drive power connector, in that pin 1 is +12v (unused), pins 2-3 are 0v (GND, Black cable), and Pin 4 is +5v (Red cable). There is no power supplied through the1MHz bus, so this connection is needed.

There is one jumper on the IDE interface – JP1, which when connected, supplies 5v through pin 20 of the interface to power CF adaptors that can use this facility. The CF adaptor supplied uses this, so leave this jumper on. The ACT light shows when the drive is in use, and will have been wired to a blue LED on the keyboard.

The IDE interface is 8 bit, as opposed to 16 bit that you would find in an Arc, or your PC. We're forced to using only 8 bit, because that is the width of the data bus in the Beeb, and what all the operating systems have been written for. There is a 16 bit interface available for the Beeb, but because there's no support software for it, it still has the same limitations as this 8 bit one. However, 1GB CF Cards are not at all expensive nowadays, and they will format to the full 512MB that ADFS (and many other computers of that era) can handle, so giving you plenty of space to work

with. A 2GB card will format to 2 x 512MB partitions – this is the maximum a BBC can handle using one Compact Flash card.

Some cards are 5v only, some 3.3v only, and some can use both. The power jumper on the CF Interface card – JP2 – will have been set to match the CF card supplied. If you use your own card, and encounter errors, try putting it to 5v. If this does not work either, then the card may not be compatible, please contact RetroClinic in this instance to see if we can shed light on the matter.

Software

Internally, the machine will contain 2 or 3 ROMs, and one RAM chip. One ROM is the Operating system, OS 1.20, another is a ROM containing 4 utilities (not fitted in all machines). That last is a Combination ROM, which contains 5 ROM images (only 4 usable at any one time, selectable by software). The main PCB will have been modified to increase the number of available Sideways Banks from 4 to 10. Two Banks are taken up by either Sideways RAM, or Sideways FRAM (see below), at locations 15 and 11. 4 Banks are in the combination ROM, and because of the way it is programmed, there are actually 5 ROMs in there. The other free socket is available for a normal BBC ROM, should you want to fit one. Because of the modifications to the main PCB, you cannot use a ROM expansion board with this type of machine.

A CD is also supplied which contains PC images for transferring files to and from the Beeb, via the CF card, or a serial cable.

RAM vs. FRAM

The machine may be supplied with one of two Sideways RAM options, both of which are 32K in size, comprising 2 16k sideways banks at locations 11 and 15:

RAM

This is like normal memory – you can load sideways ROMs into it, and it will work quite happily, until the power is turned off. Once the power is removed, it will forget its contents, and need to be reloaded when the power is restored.

FRAM (Ferroelectric Random Access Memory)

This type of RAM works in the same way as regular RAM, except that it remembers its contents, even with the power off. It does this without the use of a backup

battery. Machines with FRAM are supplied with a switch at the back of the machine, this is used to disable the contents of the FRAM, should a crash occur. This is described in more detail below.

Using ADFS with the CF system

So, we'll assume you have your machine set up, as described in Chapter 1, and can see the startup screen. If your machine was supplied with FRAM, there will be a disable switch at the back, make sure it is in the down position. Now type:

```
*XADFS
```

Type this in and press return, after a second or two, the machine will reset, and another line will appear at the top of the screen, saying "Advanced DFS 1.32". This is the ADFS 1.32. It uses some of the Sideways FRAM as workspace, allowing the value of PAGE to stay at &1900, making it very compatible with Games and software that require lots of memory. The disadvantage of this is that because space has been made for the workspace in the SWR bank, some commands have been removed. If you intend to do programming and manipulation of files, I recommend using the Acorn ADFS.

```
*YADFS
```

Now type this, and again, after a second or two, the machine will reset, and the message will now be "Acorn ADFS". This is the ADFS 1.33, which uses main memory for workspace, pushing PAGE up to &1F00. This will annoy some games as some of the usable memory has been taken up, but it retains all available commands.

```
*CLRSWR
```

Type this in and press return, after a second the machine will reset, and the ADFS will be gone. These commands allow you to initialise and reset either of the ADFSs available in the machine.

The FRAM Disable switch

Now because it doesn't forget it's data, it's possible to accidentally put some code in one of the SWR banks that will crash the machine. If your machine is fitted with FRAM, turning it off and on again will not clear the crash, as FRAM retains it's memory. Fortunately, that doesn't mean it's time to hire a skip, that's what the

little switch at the back is for. It's an Anti-Crash switch, and with it in the up position, the FRAM is disabled. With it down, the FRAM is active.

So, if the machine has crashed, it will usually boot up with a single flashing cursor on the top left of the screen. When that happens, flick the switch up, and do a **<CTRL>** **<BREAK>**. The machine should boot back up with Acorn 1770 DFS. Flick the switch back down again, but before we do anything else, we have to clear the fault, so you need to type:

```
*CLRSWR
```

The machine will then reset. If it doesn't, then the crash has probably occurred in the lower bank, so switch up, reset, switch down again, and this time type:

```
*CLRSWR L
```

The machine should now boot up with the FRAM active. If for some reason it doesn't, do the same as above again, but after putting the switch down, type:

```
*SWRTEST
```

This is an inbuilt routine to check both banks of the FRAM chip. If there are any errors reported, then it's possible the FRAM has failed, or there is another fault on the board. It's worth noting though, that if you run the command with the switch up, it will report a Fail for every test.

Loading the CF menu

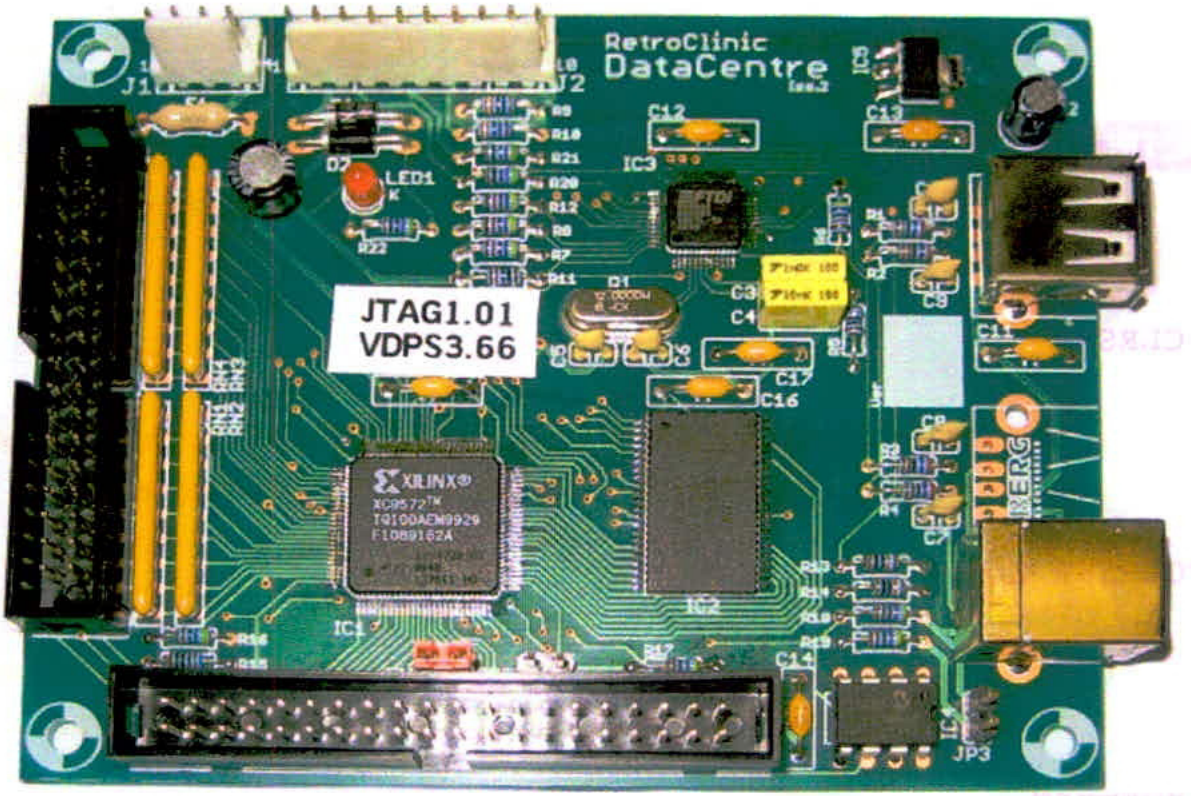
Okay, power on, little switch in the down position (if fitted), and type:

```
*XADFS
```

Wait for the machine to reboot, then hold down **<SHIFT>** and press and release **<BREAK>** while still holding down **<SHIFT>** for a moment. The machine will auto start the menu system on the CF card, and give you some choices. Follow those choices and have an explore of what's on the machine.

For more details on the ADFS system, please refer to the ADFS User Guide that came with your computer, or you may purchase one from RetroClinic.

DataCentre USB System



Usage of the DataCentre system is covered in detail in its own User Guide, which will have been supplied with the system, or is available separately from RetroClinic. The following section just gives a brief overview of the DataCentre system.

The DataCentre system has been developed over the best part of a year, and comes designed not only for ease of use for the novice user, but for maximum flexibility for those with the experience.

The philosophy behind the DataCentre system was fourfold:

1. To make the transfer of programs and data between the PC and the BBC as easy as possible.
2. To provide a USB interface that is capable of connecting to any USB device on the market, with the correct driver software.
3. To provide an upgrade path for users who have the original RetroClinic IDE CF Interface System.
4. To make a "Central Library" system, to allow users to copy, duplicate and archive their entire floppy collection.

For 1, we had examined many of the current storage solutions on the marketplace for the Beeb, and there are quite a few. Not only our own CF IDE interface, but others for MMC cards as well, of varying complexity and ease of use. One thing these all have in common, is that they need some sort of Transfer Software on the PC side, some sort of "extra step" before the data can be used on the BBC. We wanted to create a system that didn't need that extra level of complexity, something where you could simply download your data, and put it on your Beeb without any fuss or hassle of using complex and unreliable software to manipulate the images.

We think the DataCentre system has achieved that. In essence, you simply use any USB storage drive, of any capacity, be that a solid state Flash Drive, or a conventional Hard Drive in a USB Enclosure, and as long as it is formatted with the FAT or FAT32 filing system, it will be recognised. Most modern PCs, Macintoshes and Linux based machines will automatically recognise such a drive. You then download the software you want directly to it, or onto your PC hard drive and then drag the files across. Once on the USB drive, this is simply inserted into DataCentre, and the BBC can now see all those files. With a single command, the image can be imported either to one of the 4 available RAM Drives, or directly onto a Floppy Disk, if you have the appropriate hardware.

You can then go on to modify that disk image whilst on floppy or in the RAM drive, or even use your own floppy discs, and export them back to the USB drive in one of two standard universal formats, known as SSD or DSD. These images can then be immediately used on the PC by plugging the Drive back into the PC USB port, either to upload to your website, or use in an emulator, or to archive on CD etc.

The DataCentre board is also a natural upgrade path to those users who already own one of the RetroClinic CF IDE boards. The DataCentre was designed with the same connectors, so a system already fitted into a BBC can simply be unplugged, and the DataCentre inserted, and the same modified ADFS is used as you had before, to give access to your CF Hard Drive.

However, the DataCentre board IDE Interface is now a full 16 bit IDE capable port, as opposed to the 8 bit interface of the first generation IDE CF kits. This is still fully backwards compatible with the CF IDE kit, but goes on to allow you access to the full range of IDE devices available, such as PC and Archimedes Hard Drives, and also CD and DVD ROMs. With an appropriate adaptor, you can also use SATA type drives, giving access to even more modern connectivity. All these 16 bit devices will require their own driver software, as the ADFS currently available for the BBC is still only an 8 bit filing system.

Another core use of DataCentre is as a "Central Library" for all your data, and was one of the reasons the name of the board was chosen as it was. You can now take all your home brew floppy disks, and image them onto the USB flash drive with ease, using a single command. If you prefer to edit them first, you can transfer the data from floppy direct to one of the RAM drives, change what you need, then Export the data from there, so there is never any need to play with the data directly on that fragile 25 year old disk. As well as then being able to archive and back the data up quickly on a PC, or upload your entire library to your website, you now have a master archive from which to recreate floppies from your originals. No need to wear out an original disk to make copies one at a time, just use the image stored on the USB flash drive, or an image on one of the RAM drives, to make quick multiple copies of your disks. Testing with a 1772 Disk interface and a modern drive running at the default speed, you can Import or Export the entire contents of a single side of an 80 track drive in around 22 seconds, much quicker than a disc to disc copy that would be necessary otherwise.

Finally, the USB port is not just for use with USB flash drives, it is a fully featured USB 2.0 device, and can host virtually any USB peripheral you care to plug into it. As with any USB device, software is required to drive it, and this is the same for the Beeb, so don't expect to plug your DVB receiver in and get BBC 1 in Mode 7 immediately, because that's not going to happen, until someone decides to write some driver software for it. But interfacing almost any device is possible, and some

notes on the communication protocols for the USB Host device are described in more details in the DataCentre User Guide.

As well as a USB Host port, VDPS boards also have a USB slave port. This is a USB "B" type socket, and with this you can connect the DataCentre direct to your PC, using a USB A-B cable. Once connected, drivers will need to be installed, then the slave port can communicate with the PC one of two ways, either directly through API calls, or as a virtual COM port. You can also connect to HyperTerminal on the PC, and use that for bi-directional communications.

Notes

Notes

Notes

Notes

ACORN COMPUTER

RetroClinic.com
Vintage Arcade & Computing